DOCTOR OF SCIENCES

Transformation-Wise Software Architecture Framework (A Transformational Approach to Design Component Based Systems)

Gilson, Fabian

Award date:
2015

Awarding institution:
University of Namur

Transformation-Wise Software Architecture Framework

A Transformational Approach to Design Component-Based Systems

Fabian Gilson

Committee

Prof. Antoine Beugnard
External Reviewer
Telecom Bretagne

Prof. Laurence Duchien
External Reviewer
University of Lille 1

Prof. Vincent Englebert
Supervisor
University of Namur

Prof. Naji Habra
Chair
University of Namur

Prof. Patrick Heymans
Internal Reviewer
University of Namur

Thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the subject of Computer Science

Publicly defended on 2^{nd} March 2015 in Namur (Belgium)

University of Namur

PReCISE Research Center

Graphisme de couverture: © Fabian Gilson

Rempart de la Vierge, 13
B - 5000 Namur (Belgique)

Imprimé en Belgique
ISBN : 978-2-87037-872-4
Dépôt légal: D/2015/1881/12

ABSTRACT

Software architectures are meant to represent complex systems often composed by many components interrelated by different communication facilities and deployed on complex infrastructures. Architecture models depict systems at different levels of detail and must take into account multiple requirements and constraints. Without an appropriate documentation, retrieving the links between a model and its concerns may be troublesome. By loosing the design rationale, part of the architectural knowledge is lost, but recording such knowledge and the links to architectural parts is highly time consuming, even if its utility is vastly recognized.

With the multiplicity of deployment constraints, in terms of computational architectures or storage resources for example, the amount of explored alternatives also increases. Likewise, those alternatives are meaningful piece of information and are an important part of the architectural knowledge.

Many versions of a system may also co-exist and the delta between each model is sometimes arduous to identify. The traceability of evolutions between subsequent versions of a model may be useful to isolate reusable architectural patterns. The other way around, injected patterns may be scattered all over a model so that they are not recognizable anymore.

In the present thesis, we propose an architectural framework that closely relates software architectures to their requirements with their rationale, mainly in terms of design decisions. The framework relies on specific languages for component-based modeling, requirement listings and model transformations. We propose a transformation-wise approach where architectural changes are applied and documented by stepwise model transformations. These transformations play the role of traceable model evolutions that may be extracted and reproducible in different contexts, under some conditions.

Meanwhile, architecturally-significant requirements are recorded in particular listings where software engineers may refine their definitions or explore design alternative solutions. All these requirements with their decisions may be further documented with their rationale to explain the reasons why the decision has been taken, its strengths, weaknesses, hypotheses or constraints under which they have been evaluated.

RÉSUMÉ

L'architecture d'un programme informatique sert à décrire des systèmes complexes, souvent composés par de nombreux composants logiciels inter-connectés par divers moyens de communications et déployés sur des infrastructures complexes. Un modèle d'architecture permet de représenter un système à différents niveaux de détail et doit prendre en compte de multiples exigences et contraintes. Sans une documentation appropriée, retrouver les liens entre un modèle et ses objectifs peut devenir problématique. En perdant les justifications derrière un design, une partie de la connaissance architecturale est perdue. Cependant, enregistrer ce savoir et les liens entre les parties d'une architecture consomme beaucoup de temps, même si son utilité est communément reconnue.

Avec la multitude des contraintes de déploiement, en termes d'architecture physique de calcul ou de ressources de stockage, le nombre d'alternatives qui peuvent être explorées augmente. De même, ces alternatives sont d'importantes informations et représentent une partie conséquente de la connaissance architecturale.

Plusieurs versions d'un système peuvent co-exister et le delta entre chaque modèle est parfois difficilement identifiable. La traçabilité des évolutions entre versions successives d'un modèle peut être utile pour isoler des patrons architecturaux. A l'opposé, des patrons injectés peuvent être disséminés au sein d'un modèle, les rendant non-identifiables a posteriori.

Dans cette dissertation, nous proposons un cadre architectural reliant étroitement une architecture système, ses exigences avec ses principes, dont les décisions de design. Ce cadre se base sur des langages spécifiques pour modéliser des systèmes orienté-composants, une liste d'exigences et des transformations de modèles. Nous proposons une approche itérativo-transformationnelle où les changements architecturaux sont appliqués et documentés par transformations progressives. Ces transformations jouent le rôle d'évolutions traçables qui peuvent être extraites et reproduites dans divers contextes, sous certaines conditions.

Parallèlement, les exigences architecturales sont enregistrées dans des listings dédiés où les ingénieurs logiciels peuvent affiner leurs définitions ou explorer d'autres alternatives de design. Toutes ces exigences, avec leurs décisions, peuvent être justifiées plus en détail avec leurs forces, faiblesses, hypothèses ou contraintes sous lesquelles ces décisions ont été évaluées.

ACKNOWLEDGMENTS

It took some time, much more than actually planned, but I finally managed to deliver something I'm pretty happy with, even if I would have liked to explore much more areas. I wouldn't have succeeded in this adventure without the numerous people around me that supported, encouraged and suggested many comments on this personal achievement.

I first would like to thank my supervisor, Vincent, that accepted to hire me as his teaching assistant and trusted me to deviate from the initial subject to finally produce this *"baby framework"*. I enjoyed great freedom all over my Ph.D thesis, but appreciated his reviews thorough the publications and the writing of my manuscript.

I'd like to thank my accompanying committee and the members of the jury, namely Laurence, Antoine, Naji and Patrick for their valuable remarks, discussions and for their time spent at reading my manuscript. Hope they won't be the only persons who will actually read it.

I may not omit all my office mates, otherwise getting into trouble. Particularly, I express my gratitude to Nicolas, Quentin, James, Maxime, Xavier D., Gilles, Martin, Xavier L., Loup, Raimundas and also François that, at some point, gave me some time, advices, pointers, help, support or even some fun during all these years.

I also thank my close family, parents, grand-mother, brothers and family-in-law that, almost never, forgot to ask me the terrible question in the middle of a random conversation *"So, Fabian, when is it finished, your Ph.D?"*

Last but not least, I'm grateful to my beloved relatives, especially *mon Amoureuse* and *mes Petits Monstres*. I definitely wouldn't have achieved this Ph.D without my Wife that kept accurate pressure on me, and my Children that, well..., made me lose some time, but for so much fun, joy and pleasure.

CONTENTS

LIST OF FIGURES

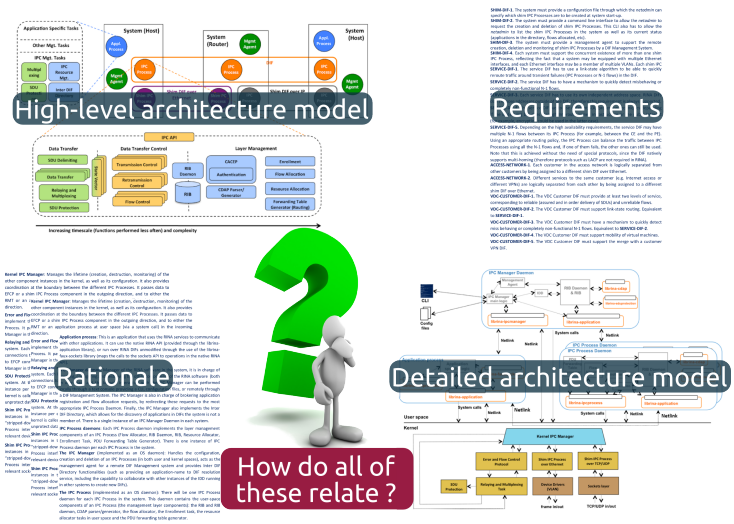
LIST OF TABLES

LIST OF LISTINGS

INTRODUCTION

Context and Problem Statement

The design and maintenance of software architectures are important challenges in the software engineering practice. For years now, systems are moving from an isolated and monolithic implementation deployed on a single platform to distributed and highly-decentralized combinations of components available on multiple technological devices. Furthermore, systems are involving heterogeneous teams that evolve in time and space. As business organizations have to cope with changes in their customers' habits, their pieces of software have to evolve too. Such maintenance and continuous evolution may lead to many problems as the system is modified by many people, but its structural representation is not updated accordingly. After some time it becomes complicated to understand the link between requirements, high-level and detailed architectural representations, and their rationale.



Problem statement (based on the IRATI project [Salvestrini et al., 2013])

A particular software system may be composed by many components, interrelated by many different communication facilities and deployed on complex target infrastructures. It is often relevant to separate between an abstract description of an

architecture, a running instance and its actual mapping onto computational nodes. Frequently, software engineers find it convenient to annotate models or architectural parts with user-defined properties to increase the understandability of models too.

An architecture model must depict a system at different levels of detail and must take into account a series of requirements and constraints. Without an appropriate documentation, the links between a model and its realized concerns are troublesome to retrieve. As we will discuss in this dissertation, many researchers have investigated different solutions to represent software architectures and trace the design decisions behind their representations. By loosing the design rationale, part of the architectural knowledge is lost, but recording such knowledge and the links to architectural parts is highly time consuming, even if its utility is vastly recognized.

With the multiplicity of deployment constraints, for example in terms of target operating systems, available computational, storage or communication resources, the quantity of alternatives explored by software designers also increases. Likewise, those alternatives are meaningful documentation support on the long run and are part of the architectural knowledge.

Many versions of a system may also co-exist and the delta between each model is sometimes arduous to isolate. Similarly, the traceability of evolutions between models may be valuable to extract potential reusable solutions *ex post*. The other way around, architectural patterns offer turnkey-style solutions that may be injected in a model to address a recurrent concern, but are sometimes diluted all over a model so that they are not recognizable anymore.

An Architectural Framework

Architectural frameworks are meant to offer a foundation to represent software architecture with the involved parties and targeted concerns [ISO/IEC/IEEE, 2011]. Many frameworks, modeling languages or documentation facilities have been proposed, we will discuss them in details in the following chapter, but they either miss important structural concepts, do not consider the relations to the requirements, omit model analysis facilities or are complicated to be understood by non-practitioners.

On the other hand, combinations of dedicated approaches and tools increase the number of models and mapping rules between these models, such that the needed documentation time grows considerably. Tracing the design decisions and rationale is a key feature, but with the time pressure on system development, it is sometimes unfeasible to invest too much time on documentation activities.

In the present thesis, we propose an architectural framework dedicated to tackle the aforesaid problems. Our framework closely relates software architectures to their requirements with their rationale, mainly in terms of design decisions, with specific modeling languages. Software architectures are produced through iterative formal transformation steps, also written in a dedicated language, close to the structural one. We call our approach **transformation-wise** because changes are applied and documented by stepwise model transformations. These transformations play the role of traceable model evolutions that may be extracted and reproducible in different contexts, under some conditions.

Meanwhile, architecturally-significant requirements are detailed in particular listings were software engineers may refine their definitions, explore alternatives, highlight conflicts or implications between identified solutions. All these requirements and decisions may be further documented directly in the listing with dedicated constructs to explain the reasons why the decision has been taken, its strengths, weaknesses, hypotheses or constraints under which it has been evaluated.

With such a **document-and-transform** method, architects, as well as other stakeholders, are able to keep a tight rein on the architecture with the evolution history of the model, together with the decision-process followed by the designers to create that particular model. Iterative design steps and evolutions are recorded as model transformations, and any decision regarding architecturally-significant requirements are also traced into specific listings, enhanced with semi-formal relationships and justifications. As we will develop in the following chapters, and as observed during an empirical study we set up, the additional work required by our stringent method is notably counter-balanced by the improvement in documentation in terms of a more systematic decomposition of requirements and recording of design rationale.

We partially built our architecture language on a preliminary work by Englebert and Vermaut [Englebert and Vermaut, 2004]. They proposed to closely relate logical and deployment architectures, annotated with *attributes*. An early attribute-driven design method was specified where components are iteratively refined into types of architectures. The initial idea was to use Colored Petri nets [Jensen, 1996] to specify the semantics of components, interfaces and connectors.

Thesis Contributions

Firstly, in this dissertation, we define an **architecture description language** that separates the definition of building blocks, that can be instantiated, concretely assembled and deployed over an abstract representation of a target infrastructure. This 3-layer language is defined upon *state-of-the-art* conceptual constructs with semantically rich connection facilities, as well as the possibility to refine their semantics with built-in or user-defined properties.

The second contribution is a **formalization of flexible component compositions**, close to the *duck typing* in programming languages, that enables to connect software components in an evolutionary manner. This connection facility also separates the concepts of connectors and communication protocols to reconcile the linkages of components from their abstract specifications to their deployment on hardware nodes.

Thirdly, **an architecturally significant requirement listing**, attached to architecture models, is proposed to record requirements with their relationships with each other, such as refinements, alternatives, mutual exclusions and so forth. This semi-formal representation also encompasses the justifications behind design decisions with dedicated rationale annotations in order to keep traces of the architectural knowledge. A **semi-formal requirement template** is also introduced to systematize the formalization of requirements.

The fourth contribution is a **specific transformation language**, working on the concrete syntax of architecture models, dedicated to modify models in a fully-traceable way. This language is able to transform a model in any way and is meant to inject new concerns or architectural patterns in a formal manner.

Based on all these contributions, a **software architecture framework** is proposed, where architecture models are iteratively transformed to implement new concerns expressed in their referenced requirement listings. This **transformation-wise** method aims at systematically record modifications in architectural models and keep a full history of explored alternatives with their rationale.

As a last contribution, a **tool support for the architecture framework** is introduced, based on the Eclipse ecosystem, to design software architectures through model transformation, record all their requirements, design rationale and explored alternatives.

Thesis Content

This dissertation is organized in seven chapters, with the following content:

In **Chapter 1**, after a brief historical overview of the software and model driven engineering fields, we first cross over the state of the art for structural and architectural modeling languages. We discuss about existing design decisions and rationale recording techniques that are relevant to our domain. We present some representative model transformation approaches from each paradigm.

In **Chapter 2**, we introduce our research questions articulated around software architecture modeling (*RQ1*), architectural knowledge (*RQ2*) and systematic iterative architectural design (*RQ3*).

In **Chapter 3**, we introduce a 3-layer architecture description language that integrates abstract building blocks definition, assemblage of running instances, deployment constraints and user-defined properties. We propose a special type of component assembly verification, close to the duck typing system. We present a case study over an online library system that we use all over this dissertation and illustrate the architectural modeling constructs with this case study.

In **Chapter 4**, we depict a semi-formal language to list architecturally significant requirements and draw relationships between them. The decision making process is captured in such listings, completed by design rationale directly inside the listing itself. We also introduce a specific template to specify requirement descriptions in a semi-formal manner. We provide some illustrations of requirement listings with the online library.

In **Chapter 5**, we discuss the advantages of a formal modification mechanism for software architectures in terms of traceability and evolutions. We define a specific concrete syntax-based transformation language. This transformation language is able to create, delete or modify any architectural element in a formal and fully traceable way. We give concrete examples of evolutions through model transformations on the online library.

In **Chapter 6**, we combine all three aforementioned languages in an architectural framework. We formalize the iterative design activity and compare it to a reference

architecture design model. We introduce the tool support for the overall framework with textual editors for all languages, a transformation engine and a Java template generator.

In **Chapter 7**, we discuss the validation possibilities we envisioned and we describe the approach we selected. We detail the protocol of the comparative case study we conducted on a class of master students at the University of Namur. We also conducted a survey to complement the case study. We discuss the results we gathered from the study, the survey and the evaluation reports written by the participants. The applicable threats to validity are finally analyzed.

The dissertation ends with a summary of the conclusions drawn thorough the various chapters, identifies the limitations of our approach and pinpoints some directions for future developments and research that worth exploring.

Related Publications

We list here the related publications that have been published or that will appear soon. When applicable, the related chapters are added below the bibliographical entry.

Conferences

Gilson, F., Englebert, V., and Matulevičius, R. (2008). A large scope transformational approach for distributed architecture design. In **Proc. of the 2nd Eur. Conf. on Software Architecture**, volume 5292/2008 of **Lecture Notes in Computer Science**, pages 330–333. Springer
[Chapter 3]

Gilson, F. and Englebert, V. (2014a). A domain specific language for stepwise design of software architectures. In **Proc. of the 2nd Int'l Conf. on Model-Driven Engineering and Software Development**, pages 67–78. SciTePress
[Chapter 5 and 6]

Gilson, F. and Englebert, V. (2015). Software architecture design by stepwise model transformations : a comparative case study. In **Proc. of the 3rd Int'l Conf. on Model-Driven Engineering and Software Development**, pages 134–145. SciTePress
[Chapter 7]

Special Issue

Gilson, F. and Englebert, V. (2014b). Transformation-wise design of software architectures. In **To appear in Communications in Computer and Information Science**. Springer
[Chapter 5, 6 and 7]

Workshops

Gilson, F. and Englebert, V. (2011b). Towards handling architecture design, variability and evolution with model transformations. In **Proc. of the 5th Workshop on Variability Modeling of Software-Intensive Systems**, pages 39–48. ACM
[Chapter 3 and 4]

Gilson, F. and Englebert, V. (2011a). Rationale, decisions and alternatives traceability for architecture design. In **Proc. of the 5th ECSA Workshop on Traceability, Dependencies and Software Architecture (TDSA'11)**. ACM
[Chapter 3 and 4]

Symposium

Gilson, F. (2010). A transformational approach for component-based distributed architectures. In **Preliminary Proc. of the MODELS 2010 Doctoral Symposium**, pages 25–30

CHAPTER 1

SOFTWARE ENGINEERING AND MODELING TECHNIQUES

To set up the scene, we present the historical context of software engineering and modeling techniques. As a starting point, we talk about object orientation and software design methods, from early top-down approaches to model driven engineering. We present a set of domain specific modeling techniques related to system architectures and deployment infrastructures. We highlight notable design documentation approaches and discuss their benefits for software development. Finally, we depict several model transformation languages and transformation-oriented tools developed in the industry and in the academic world.

1.1 A Brief Historical Overview

Early programming languages were mainly hardware-dependent. Each time a new computer architecture was released, it was accompanied with its set of programming languages. New hardwares were appearing at a regular frequency, round every one and a half or two years, asking programmers to write new programs. This production of new hardware created a profusion of languages with a rather short living time. To tackle this *write and throw-away* problem, high-level programming languages were designed to allow non hardware specialists to develop programs that could be compiled, i.e. transformed into specific machine code, for a wider range of computers.

With the emergence of high-level programming languages in the 50's and 60's, such as FORTRAN [Backus et al., 1957] and ALGOL 60 [Backus et al., 1960], the need for easy support of *component definition* [Dahl et al., 1970] has been rapidly expressed, where a program is defined by identifiable sub-programs, or *parts*, that interact via formalized operations.

1.1.1 Object Orientation Pioneer – SIMULA 67

Object orientation concepts in programming languages were formally introduced in the SIMULA 67 programming language [Dahl et al., 1970]. The authors set up the basis for a high-level programming language which was a superset of ALGOL 60. SIMULA 67, augmented version of SIMULA I at first designed as a discrete event simulation language, introduced the notions of *classes* with associated *operators*, property *inheritance* between classes and *garbage collection*.

The *outside view* of an object, making available some operations regarding this object, was separated from its *inside view*, dealing with the concrete implementation of it [Broy and Denert, 2002]. The relation between both views was formalized by Hoare's *representation function* [Hoare, 1972], also called *abstraction function*. This separation between an abstract specification and the concrete implementation made possible the inheritance mechanism where one can substitute a particular implementation to another one at compile-time for SIMULA 67.

With SIMULA 67, the ideas of encapsulation, reusability and substitutable pieces of code were born which revealed to be a major step forward for what we call now the software engineering practice.

1.1.2 Early Age of Software Engineering, Top-Down to Iterative

The word *engineering* was notably used for the first time during the 1968 NATO Conference on Software Engineering. At that time, a major *crisis* was hurting the software development industry, especially for very large systems. Projects were often overrunning their budgets and maintenance costs were very high. A common observation made by the software industry and universities at that time was the « *awareness of the rapidly increasing importance of computer software systems in many activities of society* » and that « *software engineering is in a very rudimentary stage of development as compared with the established branches of engineering* » [Naur and Randell, 1969].

Software design was mainly conceived as a linear process, with a lack of theoretical basis. However, the need for iterative design was already stressed in the late Fifties [Larman and Basili, 2003] as well as the recording of design decisions during the aforementioned conference [Naur and Randell, 1969]. One of the first recognized software development method was formalized by Benington in the mid-Fifties [Benington, 1983]. This purely linear design method consists in multiple phases from an *operational plan* where computer scientists and stakeholders define broadly the requirements of the system-to-be, to the *system evaluation* done after the complete testing in the production environment of the system.

The first reference to an iterative design process can be found in a report from Zurcher and Randell in 1968 [Zurcher and Randell, 1968]. They introduced the concept of *levels of abstraction* for a model, i.e., a set of calls between sub-components expressed in an event-based programming language, like SIMULA for example. They proposed a hierarchical view where designers refine the definition of a component iteratively by introducing new details at each step, going from a broad simulation program to a very detailed one, close to a functional prototype.

Alternatively, a first attempt to add feedback loops in the top-down process was defined by Royce in the *waterfall* method [Royce, 1970], wrongfully often known as a strictly sequential life cycle. In its work, Royce notably insisted on two important practices for software development:

- write a *preliminary program design* to highlight machine-related constraints, mainly time, storage and operational constraints;
- write a substantial and accurate documentation during the whole process and especially during the design phases.

The first recommendation was especially meaningful in its *do-it-twice* method where a first prototype of the system-to-be was developed in parallel with further system analysis and design. The pilot system could then be analyzed regarding the identified constraints so that corrective actions could be taken on the design.

The second recommendation is still relevant for nowadays systems. As we will detail in section 1.3, much research has highlighted the link between lacking documentation and problematic system maintenance. Documentation is crucial for the communication between each practitioners that appears during the system development: from requirement analysts to system designers or from system analysts to coders. Documentation is also particularly valuable for the customer that will live with the final product, since usually, a system is intended to evolve over time.

However, the waterfall model is not free of limitations. Its top-down approach is often too naive so that major high-level design rework must be done because a low-level element was not correctly specified. Second, the software evolution and components reusability is not taken into account. Third, possible discordances between the specification of a requirement and its implementation in the final product are discovered very late in the development live-cycle during the acceptance tests that are conducted at the very end of the process.

A set of alternative life-cycle models were developed to tackle the limitations of the waterfall approach. Among them, a number of iterative design methods, parents of Agile-based methods, were specified where a system is developed in successive *analysis-coding* steps. A first proposal for iterative specification and development was formulated by Edmonds when the requirements could not been fully specified beforehand [Edmonds, 1974]. In his work, Edmonds envisioned the system design through simulation models that could be validated by end-users in successive *writing* and *evaluation times* with high implication of stakeholders.

According to Larman and Basili, the first known reports on the use of iterative design methods with *feedback-driven refinements* are dated from the early seventies in the Department of Defense of the United States [Larman and Basili, 2003]. In their projects, software engineers always conducted a major analysis up-front the

iterations, with review of the produced software after each iteration. In many of these projects, from a first naive implementation of part of the system, engineers incrementally added further functionalities up to the full implementation [Basili and Turner, 1975].

A first discussion of an incremental technique for software development was published by Gilb in the book *Software metrics* where the author advocated for a design process controlled by *metrics* and introduced the term *evolution* for a software as « *a designed characteristic of a system development which involves gradual stepwise change* » [Gilb, 1977]. He suggested to implement a system in small steps to ease its evaluation and to gather user feedbacks more frequently.

These early incremental and iterative design processes differ from the ones developed from the eighties where the requirements are also analyzed, i.e. refined, during the iterations, putting the overall software development process inside iteration loops. McCracken and Jackson argued in favor of a « *heavy end-user involvement in all phases* » since even the stakeholders do not always exactly know what they want and that user needs may change during the system development [McCracken and Jackson, 1982].

Boehm defined the *spiral model* based on a repetitive development cycle without enforcing a preliminary major requirement analysis [Boehm, 1986]. Each cycle started by determining the objectives, the alternatives and constraints of (part of) the system-to-build. Then, a risk analysis is performed and strategies are identified for resolving these risks. The results of the risk resolution are then analyzed and a plan for the next cycle phase is defined. During the risk resolution phase, a prototype can be written and/or modified with the implementation of the selected objectives for the current development round. Even if this method was not the first iterative one, Boehm has been the first to formalize it and to articulate its process around risk assessments at each iteration. The software engineering practice was now moving from a top-down approach to more flexible method with frequent user-feedbacks and faster response time to changing requirements.

1.1.3 Putting More *Agility* in Software Design Methods

In his well-known « *No silver bullet* » article, Brooks, a recognized IBM software engineer, extensively praised for a switch to incremental development in order to « *grow, don't build software* » [Brooks, 1987]. At the same time, Curtis *et al.* conducted an empirical study to investigate the communication and decision-making processes in nineteen large system projects [Curtis et al., 1987]. They observed that a significant part of the success relied on a cyclic learning process, ensuring involved people shared a common vision of the system design. Iterative methods were now quitting the up-front major requirement specification phase and were incorporating refinement of their definitions in the iteration cycles.

Gilb extended his metrics-based design method in *Principles of Software Engineering Management* where he presented the *Evolutionary Project Management* (Evo) method, still articulated around quantifiable metrics, but with time-boxed iterations [Gilb, 1988]. The book presents a holistic and iterative framework capable

to deal with from-scratch or running projects, as well as for software maintenance and evolution tasks. The method is highly user- and result-centered. Each iteration step, called *evolutionary step*, is planned with, among others, focused functionalities, validation rules, constraints, assumptions, risks, costs and quantified goals that are evaluated after each step.

Among the profusion of nascent iterative approaches, the term *Rapid Application Development* (RAD) became to be extensively used by a large community of practitioners. Based on the method defined by Martin, inspired by the *Rapid Iterative Production Prototyping* method developed by Schultz, the *Dynamic System Development Method* (DSDM) was born from a small group of sixteen founding members of a devoted consortium [Larman, 2003]. This method, originally focused on software development, is now a holistic approach also used for business change management. Besides its highly prototype-oriented method, two main characteristics of DSDM are a high involvement of stakeholders during the whole project and a large decision freedom for the team members concerning the system to develop [Stapleton, 1997].

Sharing a common idea of distributed knowledge learning and decision making process as well as a time-boxed iterations, the Scrum method, originally defined for non-software goods by Takeuchi and Nonaka [Takeuchi and Nonaka, 1986], was formalized by Beedle *et al.* around *sprints*, the current iteration, *backlogs*, the work to be done, and *daily scrum-meetings* to evaluate the sprint progress [Beedle et al., 1999; Schwaber and Beedle, 2001].

A somewhat different approach was formalized by Beck [Beck, 1999]. In contrast to iterative methods where the *requirement analysis-design-code-test* tasks are organized sequentially, the *eXtreme Programming* mix all these activities and put much freedom into the programmers' hand. They decide what tasks, called *stories*, they will implement in pair, based on their own estimations and during the implementation, the customer writes the functional tests corresponding to these stories. However, XP is not the legalization of the *cowboy programming* since it requires the programmers to correctly plan and analyze the stories they are responsible to implement, so that a non trivial *forethought* activity is highly necessary.

Practitioners from, among others, DSDM, Scrum and XP communities, gathered in 2001 to produce a common set of principles in the *Agile Manifesto* [Beck et al., 2001]. The main idea of *Agile Software Development* methods is to encourage communication in teams and to lower the response time to requirement changes. However, extreme usage of Agile methods is not the panacea and most of the time, a mix between plan-driven and agile-based methods provides better success for software development [Boehm, 2002; Beck and Boehm, 2003]. Although, in several observed projects where an agile method was introduced, designers and developers even tended to produce formal documents and models [Cohn and Ford, 2003].

On the other side, the *Rational Unified Process*® (RUP), an iterative approach with extensive use of *Unified Modeling Language*™ (UML) artifacts was proposed by Rational Software [Rational Software, 1998]. In the RUP method, the software is built in four phases : *inception*, *elaboration*, *construction* and *transition* [Kroll and Kruchten, 2003]. All phases can be composed by multiple iterations and a prototype is rapidly implemented during the *elaboration* phase to define and validate the

general architecture of the system. RUP is then a kind of rapid prototyping method, but at the opposite of agile methods, with a significant use of formal documents and models.

1.1.4 Model Based Development

The usage of models probably appeared in the beginning of the XX^{th} century with the process charts specified by F.B. and L.M. Gibreth to abstractly represent business *workflows* [Gilberth and Gilberth, 1921]. The proposed formalism was designed for management purpose, i.e, *« routine of production, selling, accounting and finance »* in order to improve existing procedures.

Later, Goldstine and von Neumann defined flow diagrams to represent *code sequences*, i.e. programs [Goldstine and von Neumann, 1947]. They introduced *operation boxes* (instructions), *substitution boxes* (variable assignments), *alternative boxes* (condition branching), *remote connections* (code jump) and *induction loops*. Many flowcharting formalisms were introduced in the later years with analogous constructions [Böhm and Jacopini, 1966; Floyd, 1967; Dahl et al., 1972].

From program-centered or event-based models, the software engineering practice incorporated broader modeling techniques to represent other aspects of a system than the final code. An early top-down *command-and-conquer* approach was defined in the *Structured Analysis and Design Technique* (SADT) [Ross, 1977; Marca and McGowan, 1982]. The SADT method introduced the separation between *models*, *diagrams* and *viewpoints*. This distinction is also present in the IEEE standard for software architecture description [IEEE, 2000], that has been superseded as an ISO/IEC/IEEE standard [ISO/IEC/IEEE, 2011].

Several object-oriented modeling methods were introduced in the beginning of the nineties. The most notables of them were developed concurrently by the three researchers that wrote the specification of the UML, designed to be a general-purpose and graphical modeling language [OMG, 1997]. Rumbaugh created the *Object Modeling Technique* [Rumbaugh et al., 1990] gathering *object*, *dynamic* and *functional* diagrams. In Booch's method, six types of diagrams were defined: *object*, *class*, *state event*, *module*, *process* and *interaction diagrams* [Booch, 1991]. Jacobson, in its *Object-Oriented Software Engineering* method, added a requirement dimension with a graphical formalization for *use cases* [Jacobson et al., 1992]. Other methods were also developed at the same, but with analogous concepts, like *Object Oriented Analysis* [Coad and Yourdon, 1991], *Object Oriented Structured Analysis* [Shlaer and Mellor, 1992] and *Hierarchical Object-Oriented Design* [Delatte et al., 1992].

The first release of the UML specification in 1997 widely introduced the *separation of concerns* in a stammering software modeling community. A system can be modeled in terms of user-centric functionalities, static architectural or operable structures, interactions or collaborations between objects, and so forth. From a rather informal and ambiguous formalization of UML 1 concepts, UML 2 mainly rely on two specification documents, one to describe the language foundations (namely the *infrastructure*) [OMG, 2011c] and one for the concrete modeling constructs (namely the *superstructure*) [OMG, 2011d]. The language gained in formalization,

but defined *semantic variation points* where modelers can plug their own semantics into UML constructs, making them responsible to actually transmit correctly the model semantics to others [France et al., 2006]. UML also concretized its 4-layer meta-modeling approach making possible to define *Domain Specific Languages* (DSL) via its profiling mechanism.

The OMG proposed a general framework to enhance the separation between the core business modeling and the underlying technologies in its *Model Driven Architecture* (MDA™) approach [ORMSC, 2001]. Based on their established general purpose modeling standards, i.e, UML, *Meta Object Facility* (MOF™) [OMG, 2014b] and *Common Warehouse meta-model* (CWM) [OMG, 2003], a system can be specified in *platform independent models* separated to *platform specific models* where particular technologies are used to implement concretely the system. This separation is intended to ease the maintenance and evolution of both models separately since the business and the technology are usually not changing the same way at the same time.

From general purpose modeling languages, a recent approach consists in developing the right modeling languages for the right domain. More than just dialects or subset of general purpose languages, DSLs are usually defined as complete programming or modeling languages in order to enhance modelers and programmers productivity [Fowler, 2010]. Such languages are not a new concept. For example *Structured Query Language* (SQL) for relational databases or *Cascading Style Sheets* (CSS) for webpages style definition, are extensively used for years. Many companies also defined their own DSLs, like in the automotive industry. Besides, with the current *enthusiasm* for modeling techniques, domain specific formalisms and frameworks are being developed for a wide range of domains [Kelly and Tolvanen, 2008].

1.1.5 Model-Driven Engineering and Model Transformations

As presented in the previous section, at first, models have been used to formalize, represent and document a software. The production of lower order models or program code remained a time-consuming hand-made task which could induce translation or interpretation errors [Atkinson and Kühne, 2003; Bézivin, 2005; France and Rumpe, 2007]. From this *documentation-oriented* perspective, a current trend in modeling techniques articulates the development process around models and (formal) transformations between models. By raising the abstraction level higher, *Model Driven Engineering* (MDE) methods are trying to get closer to the system's domain model and to discharge developers from implementation or platform-related details [Kent, 2002; Bézivin et al., 2006]. Figure 1.1 shows the big picture of MDE approaches.

In a nutshell, analysts draw a model, model `A`, conforming to a given meta-model `MA`, i.e. the definition of their accepted concepts, properties and the relations between them, write a transformation in a dedicated language, also conforming to a meta-model `MT`, and produce a new model `B`. All languages meta-models must conform to a common meta-meta-model `MM` which is nothing more than the meta-

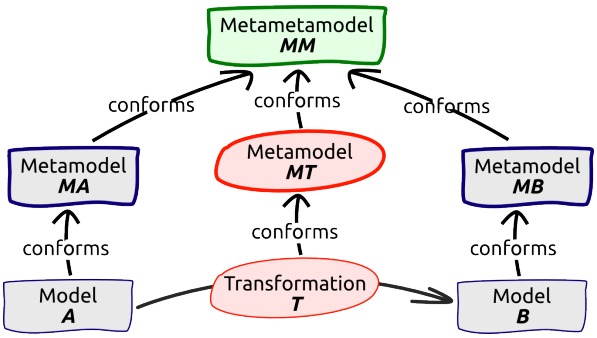


Figure 1.1: Big picture of Model Driven Engineering

model of these meta-models. There could also be more than one input and/or output models, like merging multiple models or breaking one into pieces [Kleppe et al., 2003; Mens et al., 2005; Metzger, 2005].

Models `A` and `B` can share a common meta-model, we then talk about *endogenous* transformations. On the contrary, if both models are expressed in different formalisms, the transformation is *exogenous*. As an example, the injection of a design pattern into a UML class diagram is an endogenous transformation where the production of Java code from a class diagram is an exogenous one. Alternatively, a *vertical* transformation refines a model by adding more concrete implementation details. A *horizontal* transformation, on the other hand, does not change the abstraction level neither the semantics of the model but mainly deals with model or code refactoring [Wirth, 1971].

Many MDE approaches are developed around ad-hoc modeling languages and transformation engines to address the increasing complexity of target running environments and put designers closer to the domain model [Schmidt, 2006]. In this dissertation, we will not address all these approaches since they apply to a very wide range of domains from system simulations based on the Simulink© tool[1] to generative approaches based on the UML standards.

In the present dissertation, we are mainly concerned by two aspects of software systems: **(i)** the design of the architecture taking into account the platform and deployment constraints, **(ii)** keeping explicit the links between the produced architecture models and the system's requirements and **(iii)** iteratively design a software architecture with model transformations. In the following, we will first present representative system architecture modeling approaches. Second, we discuss about existing infrastructure and deployment modeling facilities. Afterwards, in Section 1.3 we cross over existing architectural knowledge traceability techniques and their possible assets to modeling techniques. We then present the concepts behind the main model transformation languages and discuss their advantages and drawbacks.

[1] http://www.mathworks.nl/products/simulink/

1.2 Software Architecture Modeling

Several modeling languages have been proposed to model a system architecture in terms of coarse-grained *components* and the relations between them[2]. However, no real consensus exists on what should be represented in a software architecture, neither how formal the description language should be [Medvidovic and Taylor, 2000]. Many of them were academic projects and were abandoned in the early 2000 at the latest. A recent survey conducted in the industry highlighted that architects and other non-practitioners are especially interested in, among others, tool support, iterative design, model versioning and analysis [Malavolta et al., 2013].

In this section, we selected some definitions of *software architecture* (SA) and discuss them. Afterwards, we highlight the major key aspects of architecture description languages. We then present eight modeling languages before we discuss the main missing features of these modeling formalisms.

UML2 OMG component and deployment diagrams from UML2
D&C OMG Deployment and Configuration standard
SysML OMG system modeling with requirement specifications
ACME a generic architecture description language
xADL a highly extensible XML-based architecture description language
SafArchie an academic *contract-* and *aspect-oriented* language
π-ADL a formal description language for dynamic architectures
AADL an industry standard for real-time system modeling
MARTE OMG modeling standard of real-time systems
ArchiMate Open Group standard for enterprise architectures

We decided to discard several modeling languages from the present discussion, based on previous argumentation [Hilliard and Rice, 1998], comparison [Medvidovic and Taylor, 2000] and industrial survey [Malavolta et al., 2013]. We first discarded implementation specific or constraining languages, like MetaH [Vestal, 1996], UniCon [Shaw et al., 1995], ArchJava [Aldrich et al., 2002] or DAOP-ADL [Pinto et al., 2003] because we believe a platform-independent formalism is needed and more suitable, especially with the profusion of implementation platforms and technologies. Second, we do not talk over inextensible languages, such as Darwin [Magee et al., 1995] or Wright [Allen, 1997], especially for non functional properties. Third, we do not consider languages that do not have support for evolution or architectural description refinement, like Rapide [Luckham et al., 1995], because once more, evolution and iterative enrichment has been highlighted as needed features. Fourth, we do not discuss about highly domain-specific languages, such as EAST-ADL [Blom et al., 2012; EAST-ADL, 2013] or AUTOSAR [AUTOSAR, 2013a,b] since we focus on generic software architecture modeling. Last, we rejected purely enterprise architecture languages like the Zachman Framework [Zachman, 1987] or the *Architecture of Integrated Information Systems* (ARIS) [Scheer and Nüttgens, 2000] approach since they focus on business process management with insufficient support for detailed modeling of software systems.

[2] we use the *component* in its broadest sense as defined in the Oxford dictionary, as *« a part or element of a larger whole, especially a part of a machine or vehicle »*

1.2.1 A Discussion of Existing Definitions of Software Architecture

The literature, as well as the Internet, offer a wide range of definitions for a *software architecture* (SA). Across the profusion of available definitions, we selected six of them that are sufficiently concise and representative of a common vision about SA. We will discuss about these definitions afterwards.

An early very concise definition was given by Perry and Wolf where they model the software architecture as [Perry and Wolf, 1992]:

$$SA = \{ Elements, Form, Rationale \}$$

Garlan and Shaw *« treat an architecture of a specific system as a collection of computational components – or simply components – together with a description of the interactions between these components – the connectors »* [Garlan and Shaw, 1994].

Gacek *et al.* added the list of stakeholders' requirements in the definition. According to them, *« A software system architecture comprises:*

- *A collection of software and system components, connections, and constraints.*
- *A collection of system stakeholders' need statements.*
- *A rationale which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need statements. »*[Gacek et al., 1995]

As defined by Bass *et al.*, *« the software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements and the relationships among them. »* [Bass et al., 2003]

Alternatively, Kruchten stated that *« Architecture encompasses significant decisions about the following:*

- *the organization of a software system*
- *the selection of the structural elements and their interfaces by which the system is composed together with their behavior as specified in the collaboration among those elements*
- *the composition of these elements into progressively larger subsystems*
- *the architectural style that guides this organization, these elements and their interfaces, their collaboration, and their composition.*

Software architecture is concerned with not only structure and behavior, but also context: usage, functionality, performance, resilience, reuse, comprehensibility, economic and technological constraints and tradeoffs, and aesthetics. » [Kruchten, 2003]

In the ISO/IEC/IEEE standard on architecture description, the (software) architecture is the *« fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution »* [ISO/IEC/IEEE, 2011]

At our sight, the union of the definitions given by Perry and Wolf and by Gacek *et al.* supersedes all these definitions. We can represent a SA as **(i)** the **structural and behavioral** definitions of the architectural constructs, **(ii)** the **relationships** between those elements as well as the **alternative structures**, **(iii)** the **design rationale**, em-

bedding the design choices and motivations sustaining the resulting architecture and **(iv)** the **links** to stakeholders' **requirements**. Even if both definitions have been provided in the early ages of the SA discipline, many later definitions, formalisms and methods have ignored the *Rationale* dimension [Jansen and Bosch, 2005]. Furthermore, the link between an architecture and the implemented requirements is crucial to avoid knowledge evaporation [Lehman and Belady, 1985; Zdun, 2009].

1.2.2 Key Aspects of a Software Architecture Language

The software architecture discipline emerged among other, to reduce the design and maintenance costs of software systems by increasing reusability and substitutability of architectural elements [Perry and Wolf, 1992; Garlan and Shaw, 1994]. Practitioners had to move their focus from program-related concerns to architectural design since systems were becoming bigger and more complex. New methodological tools and frameworks were then needed to assist developers in their tasks. A set of specific languages were defined in the nineties mainly articulated around *components*, *connectors*, *interfaces* and *interconnected structures*.

Shaw and Garlan characterized seven levels of « *specification power* » for software architecture languages [Shaw and Garlan, 1995]:

capture	to represent a definition
construction	to assemble constituent parts to build an instance
composition	to combine multiple instances
selection	to guide designers in their choices
verification	to evaluate an implementation regarding its specifications
analysis	to measure the impact of the specification
automation	to construct instances from external properties

The authors observed that only a few studied languages offered mechanisms beyond structural definition, construction and composition. They also stressed the need for analysis and evolution capabilities in architecture descriptions.

Hilliard and Rice evaluated the expressiveness of some *Architecture Description Languages* (ADL) [Hilliard and Rice, 1998]. They described a set of « *expressive challenges* » for architectural languages including multiple viewpoints for models, design alternatives and decisions traceability, quantifiable property definitions and support for architectural pattern and styles.

Medvidovic and Taylor introduced a classification framework in which they compared a significant number of ADLs [Medvidovic and Taylor, 2000]. In their framework, they highlighted a list of needed features for such languages around three categories: *modeling components*, *modeling connectors* and *modeling configurations*. They evaluated the expressive power of the language constructs as well as their configuration abilities, like among others, scalability, model refinement, evolution support or non-functional properties. As a result of their field study, they highlighted that few of the investigated languages offered mechanisms to specify non-functional properties and to define architectural refinements. More surprising to them was the inconsistency in the definition of connectors that were often reduced to a simple semantic-less link between two components. The authors argued

for richer connector specifications and the profusion of available communication protocols and technologies nowadays underpins this claim [Beugnard et al., 1999].

Malavolta *et al.* conducted a questionnaire- and interviews-based survey on 48 industrial and academic practitioners over 40 IT companies [Malavolta et al., 2013]. Concurrently to open questions regarding the needed features of *architectural languages* (ALs), they asked the participants to rank the facilities provided by existing modeling approaches, such as tool, life-cycle wide, multiple view and styles/pattern supports. Their survey was articulated around these two research questions:

- « *What are the architectural description needs of practitioners?* »
- « *What features typically supported by existing ALs are useful (or not useful) for the software industry?* »

At the end of the survey, they highlighted nine main findings:

- most used *architectural languages* (ALs) have an industrial origin;
- ALs should support both effective communication and methodical design;
- participants prefer semi-formal and generic approaches than formal;
- tool support should provide collaboration and flexible architecture design;
- complex ALs discourage their usage (perceived high learning curve);
- adoption of an AL depends on tool and community supports;
- design, communication, analysis and link to requirements are the prior needs;
- and a ranking of the available features offered by their familiar ALs.

As already stated in academic research and comparisons of existing approaches, the penultimate recommendation summarizes adequately the major key aspects of an effective software architecture language: **(i)** it should offer appropriate **design facilities** to iteratively model a software architecture from different viewpoints and versions, **(ii)** it should be an effective **communication mean** between practitioners and stakeholders regarding structural and behavioral aspects of a system, **(iii)** it should enable some form of **model analysis** and **(iv)** it should provide explicit **linkage to requirements** with design rationale and decisions traceability.

1.2.3 UML™ Component and Deployment Diagram

Among the Object Management Group (OMG) *Unified Modeling Language* (UML) diagrams, a set of dedicated constructs have been defined to represent the structure of software systems with components [OMG, 2011d]. From an *implementation artifact* perspective in UML 1.4 [OMG, 2001], the abstraction level of components has been raised to the architecture level in UML 2 to stick to the component-based development methodologies. Figure 1.2 shows a sample `component` diagram[3].

`Components` specify modular, replaceable and refinable units with well-defined `Interfaces`. `Interfaces` are expressed in terms of properties and features. They are even provided or required by a `Component` and the set of `Interfaces` owned by a `Component` is called a contract. Since the semantics of `Components` and contracts are loose, logical as well as physical `Components` can be represented in UML 2. Connections between `Components` are defined through `Connectors`. Those ones are

[3] All diagrams presented in this section are inspired and adapted from the UML 2.4.1 specification document [OMG, 2011d]

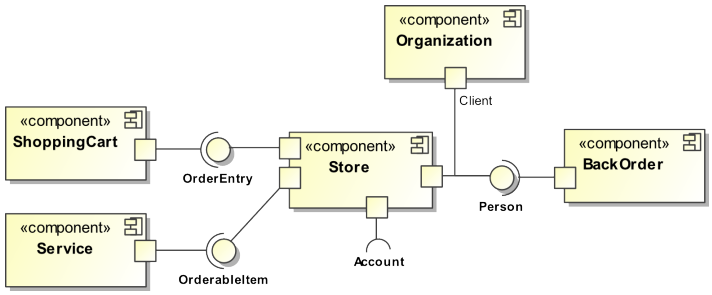


Figure 1.2: Sample `component` diagram

either `delegations`, i.e. binding contracts of the same polarity from the externally visible port to its implementation; or `assemblies` of multiple ports of opposite polarities. Assembly connectors are not necessary binary, as the *Person* interface provided by the *Store* and the *Organization* `components`. Note that the latter provides a *Client* interface which is a subtype of *Person*. `Connector` semantics can be refined by a set of constraints, expressed in *Object Constraint Language* OCL [OMG, 2012c], text, or any other formalism, and have its own behavioral specifications.

A `Component` can contain other `Components` to give a more detailed *white-box* view. In Figure 1.3, the use of `delegation` `Connectors` is illustrated by showing the internal representation of the *Store* `Component` from Figure 1.2.

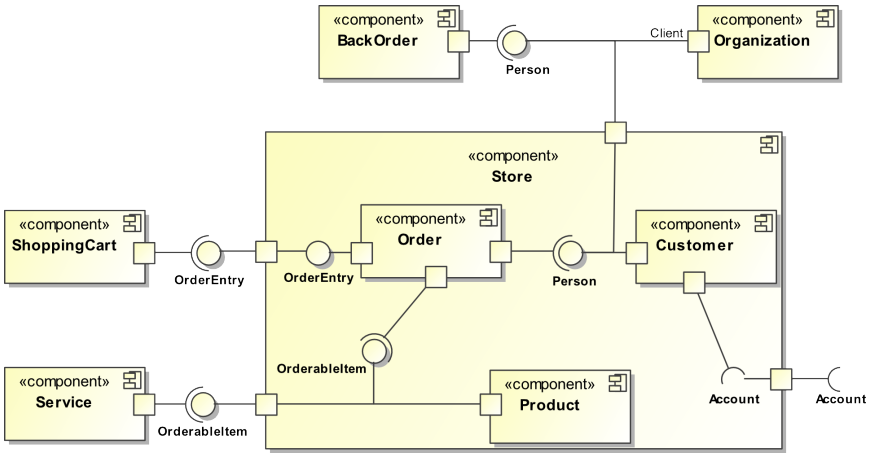


Figure 1.3: Sample nested `component` diagram

A `component` can be *realized* by a set of UML `Classifiers`, i.e. `Classes`, `Interfaces`, `Datatypes`, etc., that effectively implement the contract owned by this `Component`. The `Component` represents then an abstraction for all these concrete `Classifiers`. This flexibility offers to modelers a wide range of freedom since `Classifiers` can be specified by structural or behavioral features. The component semantics can be expressed in many different ways from statically defined

Interfaces with operations and parameters to more formal activity diagrams or state machines constrained by OCL statements for pre- and post-conditions of exposed features.

Usually, Connectors are linked to Components through Ports, but this is not mandatory by the UML specification. Port has been introduced in version 2 to refine the semantics of interaction points from Classifiers. Ports are associated to Interfaces and can redefine other Ports, again to offer the possibility of further semantics refinement from coarse to fined-grained component models.

Components can also be linked to implementation Artifacts. Artifacts are part of the deployment package of UML [OMG, 2011d] dedicated to describe execution platforms and depict any physical piece of information used or produced during a software development process. Examples of Artifacts include models, source code, database tables or deployment descriptors.

Deployment diagrams allow modelers to represent a target physical architecture with execution parameters. Such diagrams are mainly composed by Nodes and CommunicationPaths. Nodes are possibly nested computational resources that may host executable artifacts. Nodes can be specialized as Devices or ExecutionEnvironments. A Device represents a concrete physical Node and an ExecutionEnvironment refers to an execution software platform. The precise semantics of ExecutionEnvironments may be further refined by user-defined UML profiles. A CommunicationPath is simply the link between two Nodes that enables the communication between them. These are nothing more than UML Associations without any dedicated physical properties, such as communication bandwidth for example.

Artifacts may be *deployed* on Nodes under DeploymentSpecifications that specify execution parameters for these Artifacts. The exact semantics of these specifications is also intended to be further specialized in a user-defined profile. In a similar fashion, modeling elements are concretely *rendered* by Artifacts by Manifestation associations. Figure 1.4 illustrates a simple deployment configuration with three Devices, one containing an ExecutionEnvironment and one abstract Node. Also, the diagram shows the Manifestation of the *Service* Component.

Plenty of commercial and open source modeling tools exist on the market. All these tools do not have the same capabilities and do not allow to draw all types of diagrams. Also, some of them do not fully implement the standard for the models they support.

With UML 2, component diagrams became an effective component-based modeling formalism. The possibilities to refine the semantics of, among others, Interfaces, Operations and Connectors with constraints and behavioral specifications is a valuable assets for architecture modelers. However, even if the component diagrams may represent systems at different abstraction levels, the refinement mechanism is only possible via *Realization* relations which are also used to link a Component to any other UML Classifier, i.e Components, Classes, Interfaces and Datatypes. It is then difficult to know if the target realization is an abstract semantics refinement or a concrete system architecture.

On the other hand, deployment diagrams are rather simplistic, especially to

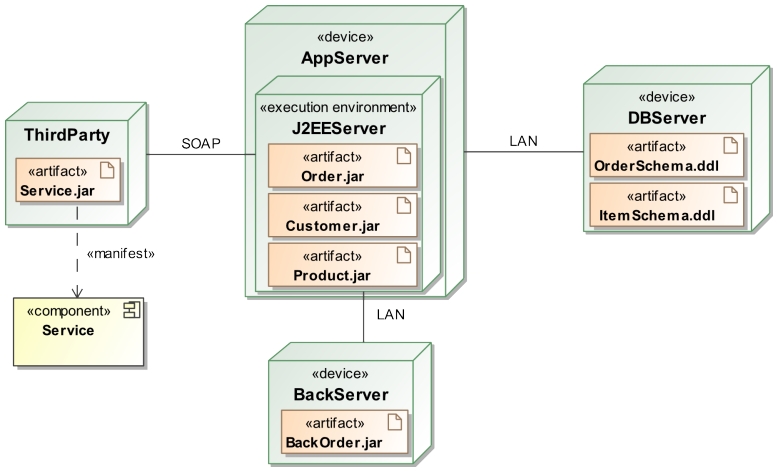


Figure 1.4: Sample deployment diagram

model communication facilities between `Nodes`. Deployment diagrams focus mainly on the static affectation of artifacts onto computational nodes. The way to define communication, processing, memory space or runtime properties is left up to the user to specialize the deployment constructs by UML profiles. Also, the graphical notation is quite poor, since the distinction between devices, databases or runtime environments can only be expressed by stereotypes.

1.2.4 OMG Deployment and Configuration Standard

The OMG wrote a dedicated standard for the *Deployment and Configuration of Component-based Distributed Applications* (D&C) [OMG, 2006]. This standard focuses on the deployment of complete software systems and is structured around three dedicated models and one deployment process. All models are augmented by management interfaces with predefined operations to manipulate the created models and artifacts metadata. The general idea relies on the *Model-Driven Architecture* (MDA) approach that separates *platform independent* and *platform specific models*. The standard defines also a UML 2 profile for all structural modeling concepts.

The *Platform Independent Model* (PIM) describes components somehow differently than UML component diagrams in order to be applicable to a wide range of component-based modeling language. In a *Component Data Model*, the logical architecture is mainly depicted in terms of components, interfaces and ports. Interfaces and ports are only referenced by their names. Nested components are expressed by `ComponentAssemblyDescriptions` and allow to define complex white box views of a particular implementation. Connections between ports are done in a broadcast way analogously to an electrical circuit. Many description constructs are available to specify deployment constraints, like the `Locality` saying if a component must be isolated onto a physical node, or whether the component satisfies a requirement.

The physical infrastructure, called the `Domain`, is described in a *Target Data Model*, which is close to the UML deployment diagram. Basically, `Nodes` are directly bound to each other through `Interconnects` or indirectly via communication `Bridges`. Nodes can have access to `SharedResources`, which can be themselves `Nodes`, `Interconnects` or `Bridges`.

The deployment planning is modeled in an *Execution Data Model*. The `DeploymentPlan` describes the artifacts that will be deployed and how they will be instantiated. A `DeploymentPlan` is nothing more than a flattened aggregated `ComponentAssemblyDescription` of the overall system with additional properties. Other common elements to all model types can also be defined, such as `Properties`, `DataTypes` and `Requirements`. The latter is used to link implementation or assembly descriptions to their fulfilled requirements, but remains semantics-less in the standard so that any type of requirement can be expressed.

The standard describes a set of abstract `Actors` that may intervene in the implementation and deployment phases of a software system. These persons are in charge of writing the various *description* documents and are divided into *development* and *deployment* `Actors`.

- Development `Actors`

`Specifier`	writes the interface specification
`Developer`	implements a component interface
`Assembler`	reuses existing components to satisfy an interface
`Packager`	packages various implementations of the same interfaces
`Domain admin`	models the target `Domain`

- Deployment `Actors`

`Repository admin`	installs a package into a repository
`Planner`	defines and plans the deployment of a software
`Executor`	supports the software release phase

The `Planner` is the key actor of the D&C standard. He is in charge of finding valid deployment configurations of the software and checks whether the requirements are actually fulfilled by the selected components, connectors and resources.

To effectively start the deployment process, the software must have been packaged according to required metadata. This process is divided in five phases:

Installation	put the packages into a repository
Configuration	specify the runtime options of the software
Planning	decide how, when and where the software will run
Preparation	any needed conditional actions on the runtime system
Launch	components instantiation and runtime configuration

The standard finally describes the mapping from their PIM models to the *Common Object Request Broker Architecture* (CORBA®) Component Model (CCM) [OMG, 2012b]. The data models are either mapped to *Extendable Markup Language* (XML) schemas [W3C, 2012a,b] or to CORBA *Interface Description Language* (IDL) data structures [OMG, 2012a]. All management interfaces defined alongside the data models are translated into IDL interfaces.

This document specifies generic objects and is particularly dedicated to support automated (re-) deployment of a software application. However, the semantics of

many structural constructs is very poor (especially the `Requirement` which has no semantics description at all). It also relies a lot on textual and untyped descriptions for the usage, the assemblage or even the ports of components. The modeling of communication paths with `Bridges` to which no properties can be added is not really useful and concrete binding between distributed components is not trivial in many cases where routers or firewalls are in the middle. Furthermore, the concept of `SharedResource` lacks of properties since, for instance, using a network shared drive implies different constraints than requiring a local shared memory space for two applications running on the same machine. Besides, dynamic or incremental development and deployment of components is not supported.

To the best of our knowledge, the *Deployment And Configuration Engine* (DAnCE) is the only D&C compliant tool [Deng et al., 2005]. This academic tool focuses on the deployment of real-time and embedded systems based on *Quality of Services* concerns.

Upon its advantages of standardizing the deployment of distributed component-based applications, D&C is not really suitable to model a target infrastructure during the software development process for *in-development* physical constraints verifications. Also, the mapping from PIM to PSM is sufficiently adequate with the *Corba Component Model*, but no further proof is given regarding other existing component-based technologies or features.

1.2.5 OMG System Modeling Language™

The OMG *System Modeling Language* (SysML) is a UML 2 profile intended to describe systems applications [OMG, 2012d]. SysML focuses on any engineering system that embed pieces of software but is flexible enough to depict hardware, processes, information or even personnel systems. SysML reuses part of UML 2 constructs and provides additional features, such as requirement-related elements. The main evolutions from the first version [OMG, 2007] until the actual one include primarily alignments to changes in UML semantics and some improvements on the definition of units of measure. Although, the most significant change was released in the last version and concerned the definition of ports which now enable to represent hardware connections on systems in an easy and straightforward way with typed flows of *"objects"*.

The standard is divided in three different types of constructs: *structural*, *cross-cutting* and *behavioral*. The basic SysML structural element is called `block` and is defined as a collection of structural and behavioral features that are exposed to the outside. SysML structural models are either representing a *black-box* view, namely *Block Definition Diagram* (BDD) or a *white-box* view, the *Internal Block Diagram* (IBD). A BDD represents the externally visible features of `blocks` with relations between them. An example is depicted in Figure 1.5 where a high-level decomposition of a *hybrid car* system is presented (without `block` properties)[4].

[4] All models presented in this section are inspired from the SysML 1.3 standard [OMG, 2012d]. All models are mostly *hardware-oriented*, but SysML can obviously represent software components.

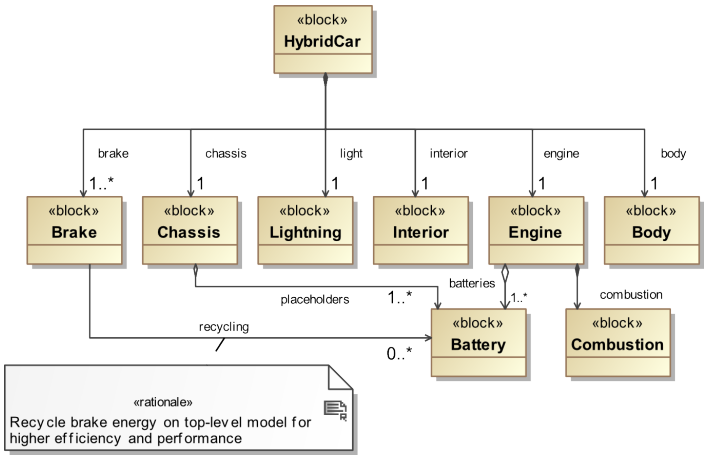


Figure 1.5: Sample BDD of an *HybridCar* system

The diagram presents a hierarchical view of the whole *HybridCar* system. It is composed by six parts, depicted by *black diamond* links which are the direct *children* `blocks` from *Brake* to *Body*. This `PartAssociation` is equivalent to a `composition` in UML 2 class diagram. At the opposite, the *Battery* is *shared* between the *Chassis* and *Engine*. A first usage of a `rationale` comment is also presented in the model. Actually, SysML allows to add extra free-format details on any modeling element, even on relations, to further detail a decision regarding, for example, the design or requirements, as we will discuss later in this section.

Structural `blocks` are defined by `Properties`. A `property` can express a usage or role that plays a `block` within the context of its parent `block`. The precise semantics of a `property` is defined through its type. First, it can be an attribute to which values can be assigned. These values can depend on the usage of the `block` as a `part` of another `block` in a specific context (a `part` is then a type of property). This mechanism allows modelers to specify one type of `block` that can be aggregated in different `blocks` with different values. Second, `blocks` can also contain constraints that are typed by special `constraint blocks`. The third type of properties are references to other blocks, either with UML-like class associations, or fully-typed interactions through `Ports`. `Ports` can be specialized as `proxy` or `full ports`[5]. A `proxy` exposes (part of) the features of a `block` or the ones of its `parts`. It does not express behavioral feature by itself, but exposes the ones defined in its typing `InterfaceBlock` gathering some features that are owned by other `blocks`. At the opposite, a `full port` exposes features that it handles by itself, thus that are not owned by its parent `block`. Figure 1.6 shows the type of the *ICombustion* port (that will be used in Figure 1.8). Concretely, this `port` type is a `block` with four value

[5]This specialization, i.e. stereotype, is left out to the modeler, depending on its freewill, even if the standard identifies only these two *usage patterns*. Additionally, since this specialization appeared in the last version of SysML, the standard does not constraint to choose between both types for backward compatibility reasons.

properties and two operations. Another way of typing a connector between blocks is by flow properties that represent a single item that may *flow* from one block to another. Figure 1.7 illustrates the *Transmission* block of the car system with three different flows.

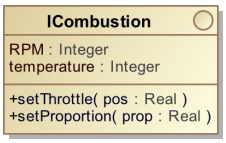


Figure 1.6: Definition of the *ICombustion* port type

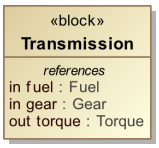


Figure 1.7: Flow properties of the *Transmission* block

Aside these *black-box* views, *Internal Block Diagrams* are meant to depict the *white-box* structure and concrete relations between the properties that compose a block. Each block defined in a BDD can be refined into an IBD containing other blocks that can be refined again, and so forth. The refinement process ends when an IBD contains only parts, i.e. block *instances*, with their relations. This technique allows SysML modelers to represent systems at diverse levels of abstraction. Figure 1.8 presents the internal structure of the *Engine* and illustrates the usage of simple associations, typed ports and flows[6].

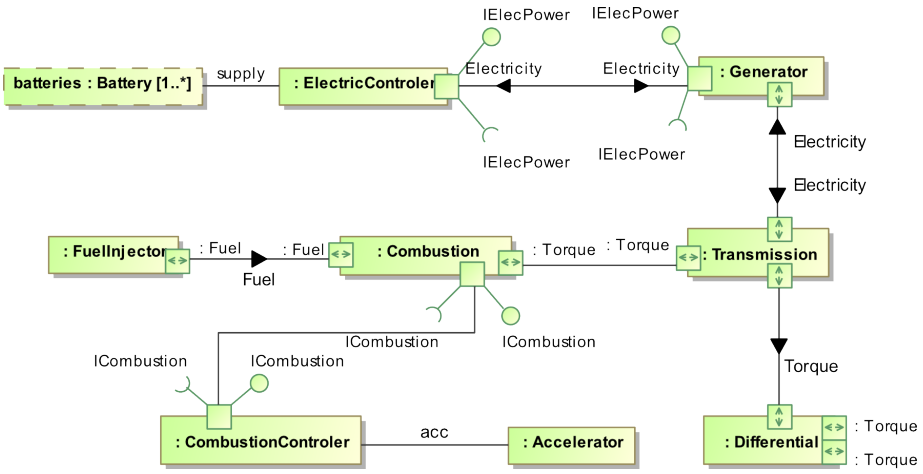


Figure 1.8: Sample IBD of the *PowerSubsystem*

Full and proxy ports are represented by a *ball-and-socket* notation and flow ports with angle brackets. One block, the *Battery* has been represented with

[6]According to the examples in the specification, the *ball-and-socket* elements from a port type should have different names of the form ICombustionCmd and ICombustionData, to express the direction of the call from the *command* to the *data*. However, the modeling tool we used did not enable us to differentiate the names of both parts of the same interface.

dotted box to denote the « *use-not-composition* » relation as presented in Figure 1.5, meaning that the *Engine* uses it but does not own it.

Next to structural models, system `requirements` are listed in a requirement table and can be represented in dedicated models to express:

refinement a use case is refined into a full text `requirement`
derivation a lower order `requirement` is derived from a higher order one
satisfaction a `requirement` is satisfied by a `block`
verification a `testcase` verifies the completeness of a `requirement`
documentation traces `rationale` and `problems` between `requirements`

Figure 1.9 illustrates the requirement model for the *Master Cylinder Safety* (id.*1*) and *Reuse Brake Energy* (id.*2*) `requirements`. From the *Decelerate Car* use case, more complete descriptions of `requirements` are derived. The first `requirement` is the union of two higher order and less descriptive `requirements` (with id.*1.1* and *1.2*). A `rationale` is attached to `requirement` with id.*2* to further justify the design choices and express the reason why the `requirement` is actually satisfied. Last, `testcases` are linked to the *Separate Reservoir* `requirement` to verify that it is actually implemented.

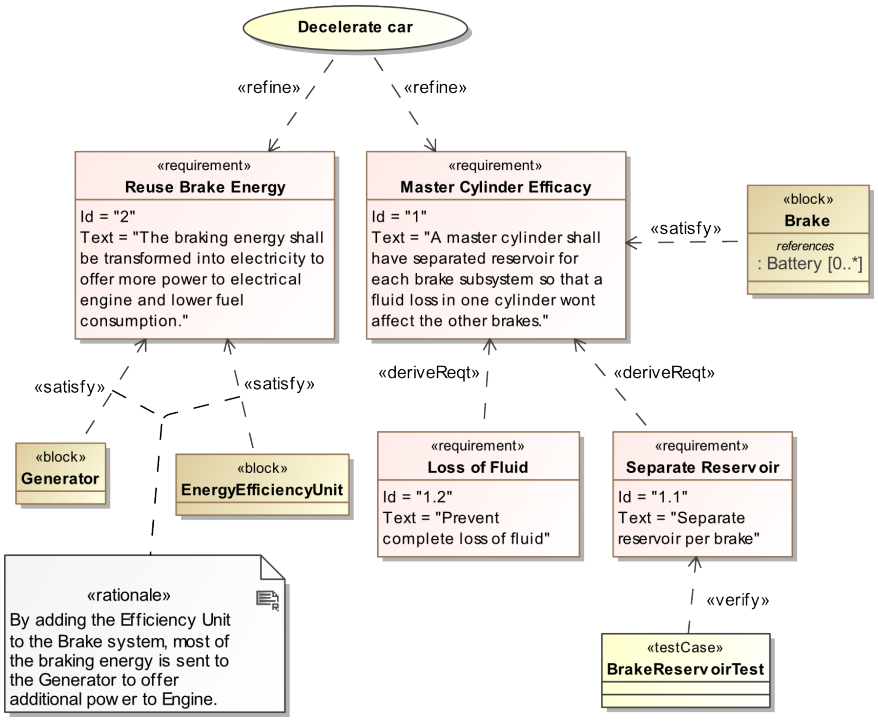


Figure 1.9: Sample Requirement diagram of the hybrid car

Modelers can create domain specific `viewpoints`. A `viewpoint` is designed to abstract particular aspects of the system for identified stakeholders. A `view` is then the subsystem that conforms to a `viewpoint`.

The system behavior can be specified by specialized UML 2 activity diagrams, interaction diagrams and state machines. Activity diagrams have been augmented to define special activities to match the semantics of SysML `ports`. Timing, communication and interaction overview diagrams have been excluded from the SysML profile. UML2 state machines can be used as-is. All these models enable semi-formal specifications for the concrete interactions of `blocks` and `ports`.

The `block` construct is a very flexible mechanism to define a wide range of structural modeling elements. The `ports` and the type of connectors complete this flexibility since software interface-based communication as well as flows of tangible objects can be exchanged between blocks. Besides, the BDD-IBD semantics refinement mechanism, coupled to domain specific `views`, enable layered representations of systems depending on the desired abstraction level and stakeholder `viewpoint`. Linkage from requirements to model elements and the possibility to add design rationale to models are also interesting assets of the SysML standard.

Re-using UML 2 behavioral specification mechanism is a relatively effective way of communication to non practitioners since the syntax and semantics are easy to learn. Also, UML 2 is somehow used in the industry [Malavolta et al., 2013] and sequence diagrams and state charts are one of the most popular diagrams [Petre, 2013], so the learning curve of SysML behavioral aspects should be negligible for UML practitioners.

However, one of the goal of SysML is to support the development of engineering systems including avionics and automotive. `Constraints` can be used to verify some properties of a system, but behavioral aspects cannot be validated because no formal notation is provided. Furthermore, the extensive flexibility of `blocks` and `ports` is quite complex to get familiar with at first sight. Patterns or styles definition and usage in SysML models is quite complex, again because of the considerable flexibility of structural constructs. Coupled to the nested BDD-IBD mechanism, it would have been interesting to define a versioning and linkage mechanism between related models instead of depending on the model description or rationale properties.

1.2.6 ACME - Architectural Description of Component-Based Systems

ACME has been designed as a *generalization* of other existing *Architecture Description Language* (ADL) [Garlan et al., 1997, 2000]. It mainly deals with the static structure of an architecture and it does not provide any behavioral mechanism. However it is intended to be extensible by user-defined properties.

ACME contains seven core concepts:

`component`	computation or data store element
`port`	interface point of `Component`
`connector`	communication mediator
`role`	interface point of a `Connector`
`representation`	internal description of a `Component`
`system`	configuration of `Components` and `Connectors`
`bindings`	mapping between internal representation and external interface

Because ACME is intended to be generic, any definition of an ACME model element can be refined by user-defined properties. A `property` has a name, an optional type and a value. The type must be either primitive, like *integer* or *string*, or complex, like *sets*, *records* or *lists*. The core ACME elements are illustrated with a simple Client-Server sample in Listing 1.1[7].

```
system simple_cs = {
  component client = { // client definition with one port
    port send-request;
    property source-code = "CODE-LIB/client.c";
  };
  component server = { // server definition with one port
    port receive-request;
    property max-concurrent-clients : integer = 1;
  };
  connector rpc = { // connector with two roles
    role caller;
    role callee;
    property synchronous : boolean = true;
  };
  // binding between the client and the server via the connector
  attachment client.send-request to rpc.caller;
  attachment server.receive-request to rpc.callee;
}
```

Listing 1.1: A Client-Server sample in ACME

ACME does not process property-related values. Since its is somehow designed as an *interchange description language*, it is up to the implementation tool to make use of these properties. Also, the definition of the concrete semantics of such properties remains in the users' hands.

An ad-hoc constraint language has been added to ACME to specify logical predicates and primitive functions over architectural specifications [Garlan et al., 2000]. Usual *First Order Predicate Logic* (FOPL) operators are available, such as conjunction, implication or quantification. Furthermore, designers can check for the existence of a property or retrieve the architectural elements of a lower order model element with predefined ACME functions like the `connectors` of a `system` or the `roles` of a `connector`. The constraints mechanism is illustrated in Listing 1.2.

```
  connector MessagePath = {
    role source;
    role sink;
    property bufferSize : int;
    property expectedThroughput : float = 512;
    rule bufferLimit = invariant (bufferSize >= 512) and (bufferSize <= 4096);
    rule expectedTP = heuristic expectedThroughput <= (queueBufferSize / 2);
  }
```

Listing 1.2: Simple constraints in ACME

Two types of constraints can be defined: `invariant` and `heuristic`. An `invariant` is a rule regarding a property that may never be violated. The `heuristic` is an observed rule that may be violated in specific cases. When writing constraints

[7] All examples have been adapted from [Garlan et al., 1997, 2000] or from the ACME website according to the latest release of ACME studio. The grammar is available at http://www.cs.cmu.edu/~acme.

inside an element, references to that element are made by the use of the `self` keyword. In the above listing, the *MessagePath* **must** have a *bufferLimit* between 512 and 4096 (bits). It also **may** have an *expectedThroughput* lesser than the half of its *queueBufferSize*.

ACME provides a way to define architectural styles. Taylor *et al.*, define architectural style as *« a named collection of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system »* [Taylor et al., 2009].

Two mechanisms are available in ACME. First, an architect can specify a structural type for `components`, `connectors`, `ports` and `roles`. A `type` is defined exactly the same way as another construct, with properties and/or constraints. It can be reused in any `system` definition. Second, an overall `system` can be defined as a `family` (the ACME keyword for a style) to constraint a particular architectural structure and make it reusable. ACME provides type-checking facilities in order to ensure a system respects the definition of its possibly multiple types. Both concepts are illustrated in Listing 1.3.

```
family PipeFilterFam = {
  component type tilterT = {
    port stdin;
    port stdout;
    property throughput : int;
  };
  component type UnixFilterT extends FilterT with {
    port stderr;
    property implementationFile : String;
  };
  connector type PipeT = {
    role source;
    role sink;
    property bufferSize : int;
  };
  property type StringMsgFormatT = record [size:int; msg:String;];
  rule typecheck = invariant forall c in self.connectors | HasType(c, PipeT);
}

system simplePF : PipeFilterFam = {
  component smooth : FilterT = new FilterT;
  component detectErrors : FilterT = new FilterT;
  component showTracks : UnixFilterT = new UnixFilterT extends with {
    property implementationFile : String ="IMPL_HOME/showTracks.c";
  };

  connector firstPipe : PipeT;
  connector secondPipe : PipeT;

  attachment smooth.stdout to firstPipe.source;
  attachment detectErrors.stdin to firstPipe.sink;
  attachment detectErrors.stdout to secondPipe.source;
  attachment showTracks.stdin to secondPipe.sink;
}
```

Listing 1.3: Architectural style in ACME

In the example above, a *pipe and filter* architectural style is defined with two types of `components`, one type of `connector` and some related properties and constraints.

This style is *instanciated* in a 3-`Components` model where the concrete `attachments` between the `components` are specified. Note that the example above shows also the extension facility, i.e. component inheritance, where the `UnixFilterT` extends the `FilterT` with an additional port and specific properties.

In ACME, a `component` can be refined into a `representation` where an inner `system` is specified and the linkages between the outer Ports to the inner ones are mapped in `bindings` rules. A `component` can have multiple `representations` and `representations` are recursive, i.e., a `system` defined inside a `representation` can be further refined into lower-level ones. We show an internal representation of a `Component` in Listing 1.4.

```
component theComponent = {
  port easyRequests;
  port hardRequests;
  representation {
    system details = {
      component fastButDumbComponent = { port p; };
      component slowButSmartComponent = { port p; };
    };

    bindings {
      easyRequests to fastButDumbComponent.p;
      hardRequests to slowButSmartComponent.p;
    };
  };
};
```

Listing 1.4: Representation and rep-map rules sample

Behavioral specifications are not supported by the language itself, but an extension is provided to add *Finite State Process* (FSP) definitions to components. The FSP specification is compliant to the *Labeled Transition System Analyzer* (LTSA) of the Imperial College of London[8].

ACME is supported by a modeling tool called ACME studio[9], developed as an Eclipse [10] plugin. It provides a graphical and textual editor for ACME models and integrates the Armani [Monroe, 2001] constraint checker. It also allows designers to define their own notations for architecture models. The tool is still currently maintained[11]. It has been extended by undocumented performance and FSP analysis plugins.

However, the edition of textual models is relatively complicated since no autocompletion facility is implemented and the only available formal documentation is the Armani BNF grammar[12] which is not really readable. Hopefully, a summary of the syntax is provided on the website, but does not always respect the formal grammar. For example, the definition of `attachments` are still compliant to the initial syntax, but does not comply to the actual second version. Also, the tool is not

[8] http://www.doc.ic.ac.uk/ltsa/

[9] http://www.cs.cmu.edu/~acme/AcmeStudio/

[10] http://www.eclipse.org

[11] according to their bug tracking system at http://acme.able.cs.cmu.edu/mantis/my_view_page.php, the last bug date back to September 27^{th}, 2013

[12] http://www.cs.cmu.edu/~acme/html/ArmaniParser.html

free of bugs. Among others, the constraint validation process often keeps references to old errors and the concurrent graphical visualization and edition of the same model sometimes delete parts of a that model if some inconsistencies arise.

ACME offers a lightweight and extensible modeling language for software architecture. Structural elements can be refined by constraints and first order logic statements can also be expressed to define more complex conditions on properties. Types and architectural patterns can also be defined in a reusable manner through models. *Components* can also be represented at different level of abstractions through `representations`. However, ACME does not allow to define interfaces with their operations, like in many other component-based languages, with precise *object oriented method*-like semantics. `Ports` are mainly entry/exit points in `components` with no semantics at all. Connectors are limited to point-to-point bindings between components. Furthermore, no viewpoint definition mechanism is supported and neither technological nor platform-related structural elements exist for abstract deployment validation, for example.

1.2.7 A Highly Extensible Architecture Description Language

As an answer to the plethora of architectural notations, Dashofy *et al.* introduced an extensible *XML-based Architecture Description Language* (xADL) [Dashofy et al., 2001, 2002, 2005]. The main objectives of xADL is to propose a syntax-based infrastructure to provide [Dashofy et al., 2005]:

- an extension mechanism to rapidly build a custom ADL
- a reusable set of features, or *modules*, devoted to ADL development
- flexible tools to support the development and usage of the ADL

xADL is made of a collection of XML schemas addressing specific concerns such as, among others, *Structure and Types*, *Runtime Instances*, *Java Source* or *Variants* [Dashofy, 2007]. The basic primitive constructs are defined in the *Structure and Types* module which can be viewed as the structural core of the xADL language. It defines:

`Component`	computation block with a set of `interfaces`
`Connector`	communication block with `interfaces`
`Interface`	semantic-less interaction point
`Link`	connections between `interfaces`
`Subarchitecture`	composite `component` and/or `connector` topology
General group	semantic-less grouping of elements (may be extended)

Figure 1.10 illustrates a sample architecture of a TV tuner connected to an infrared receiver, its XML representation being illustrated in Listing 1.5.

Figure 1.10: Sample TV tuner architecture (from [Dashofy, 2007])

```
<xArch> <!-- the TV set architecture model -->
  <archStructure id="tvset">
    <description>TV Set</description>

    <component id="tuner"> <!-- the TV component -->
      <description>TV Tuner Component</description>
      <interface id="tuner.channel">
        <description>ChangeChannel Interface (in)</description>
      <direction>in</direction>
      </interface>
    </component>

    <component id="ir"> <!-- the infrared receiver component -->
      <description>Infrared Receiver Component</description>
      <interface id="ir.channel">
        <description>ChangeChannel Interface Tuner to Connector (out)</description>
        <direction>out</direction>
      </interface>
    </component>

    <connector id="tvconn">  <!-- the connector -->
      <description>TV Connector</description>
      <interface id="tvconn.in">
        <description>ChangeChannel Interface (in)</description>
        <direction>in</direction>
      </interface>
      <interface id="tvconn.out">
        <description>ChangeChannel Interface (out)</description>
        <direction>out</direction>
      </interface>
    </connector>

    <link id="link1"> <!-- link from TV tuner interface to connector -->
      <description>Tuner to Connector</description>
      <point> <anchor href="#tuner.channel"/> </point>
      <point> <anchor href="#tvconn.out"/> </point>
    </link>

    <link id="link2"> <!-- link from connector to infrared receiver -->
      <description>Connector to IR</description>
      <point> <anchor href="#tvconn.in"/> </point>
      <point> <anchor href="#ir.channel"/> </point>
    </link>
  </archStructure>
</xArch>
```

Listing 1.5: Sample TV tuner in xADL 2.0 XML format (from [Dashofy, 2007])

Two `components` and a `connector` are defined in the model, respectively the tv *tuner*, the *ir* receiver and the *tvconn*. Both `components` have a single `interface` of opposite polarities (called `direction`). The *tvconn* `connector` has two interfaces that will be used to link on one side the *tuner* and on the other side the `ir`. By default, any construct semantics may be refined by a description, but no details on the interface can be defined structurally. An *structural* xADL model concentrates mainly on the model topology.

The question of an appropriate and flexible tooling support is critical in xADL. Since the language needs to be extended, the tool cannot rely on neither fixed syntaxes or meta-models. To that purpose, the authors developed an extensible

framework called ArchStudio[13] as an Eclipse plugin. The collection of tools mainly consists in a *xArchADT* facade used by two editors (graphical and tree-based), an XML *DOM Implementation* [W3C, 2004] that handles extension schemas, an *Apigen* tool that generates Java interfaces from the extension schemas and a *Data Binding Library* that provides these interfaces to the facade. The current version of ArchStudio emerged from an initiative started in the 90's with the C2SADL architecture description language [Medvidovic et al., 1996, 1999] and evolved to the creation of xADL. At the opposite of ACME that has been proposed as an interchange architecture language, xADL is more an architectural framework and tooling environment to develop software architecture notations instead of being a language by itself.

Many extensions have been proposed to xADL like behavioral specifications with *statecharts* [Naslavsky et al., 2004], architectural aspects [Fuentes and Gámez, 2007] or Service-Oriented concerns [Pannok and Vatanawood, 2013]. Also, an analysis framework, namely *Archlight* [Dashofy, 2007], offers possibilities to add custom architectural *test-based* model verifications, close to model checking practices. The overall ArchStudio tooling is still under development, currently on xADL 3.0, even if no technical reports or research papers have been published later than the Ph.D dissertation of Dashovy in 2007 that relied on xADL 2.0[14].

xADL offers an interesting platform for *architecture specific* language development. It provides syntactic support to build custom viewpoints concerning a software architecture. However, except for implementation artifacts, no research to date has been made to connect xADL features to other life-cycle documents like requirement definitions. With available current extensions, design alternatives can only be represented in terms of *variants* which are alternative configurations based on *hardcoded* conditions, close to the *Product Line Architecture* [Bosch, 2000] domain. Finally, the available construct expressiveness is, by design, weak since the authors reject the responsibility of semantics definition and verification on the ADL developer.

1.2.8 SafArchie and TranSAT

SafArchie is an academic *3-layer* architecture description language designed to define separately a type of software architecture, its instance and the deployment target with validation facilities for component compositions [Barais, 2005]. The language is expended by the TranSAT transformation framework devoted to describe and inject pattern-based concerns, i.e. architectural aspects, into an existing architecture model.

SafArchie models can represent an architecture from three different viewpoints: *type* model, *logical* model and *physical* model. An architecture *type* relies on a small set of constructs, such as `primitive` and `composite components`, communication `ports`, synchronous `operations` and `links`. The *logical* model is very close to the *type* model and depicts a valid instance of its *type* model. A *physical* model

[13] http://isr.uci.edu/projects/archstudio/
[14] http://isr.uci.edu/projects/archstudio/publications.html

represents the target deployment infrastructure in terms of `site`, communication `interface`, communication `channel`, routing `node` and `route`.

SafArchie provides textual, XML and graphical notations to describe models. An example depicting a *DataPhoneCenter* type model is shown in Figure 1.11.

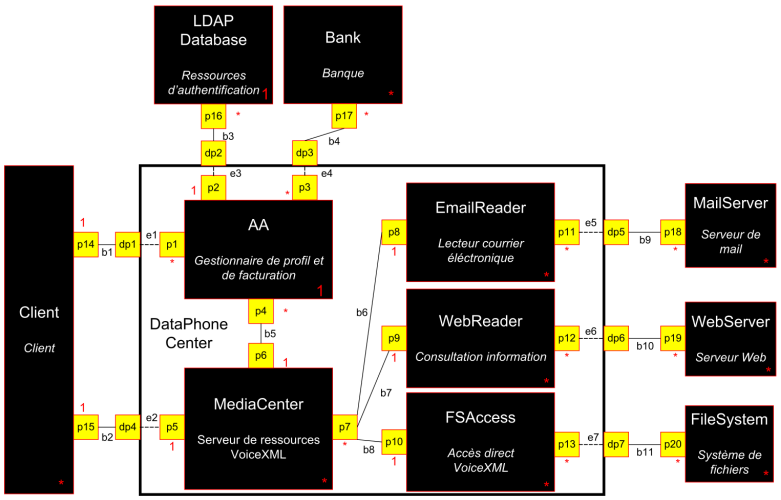


Figure 1.11: Example `type` model in SafArchie (from [Barais, 2005])

In SafArchie, in order to validate the composition of components, three types of `contracts` must be specified [Meyer, 1992; Beugnard et al., 1999]:

port define the set of operations a `port` must at least offer
assertion pre/post conditions on operations
behavioral a labeled transition system of the behavior of a component

The port contract is simply defined as possibly multiple conjunctions of available operations, of the form:

$$open + close + (read|write)$$

where $+$ denotes an unordered conjunction and the $|$ denotes alternatives. The above contract specifies that the target element must offer the *open* and *close* `operations` and either the `read` or the `write` `operation`.

Assertions are defined in an OCL dialect. For instance, the condition saying that the withdraw limit is set to 2000 in the Bank *context* accessible via `port` *p17*, is expressed by the following assertion:

$$context\ Bank :: p17 :: withdraw(amount : long) : long$$
$$pre : self.amount < 2000$$

The behavioral specification is expressed in ad-hoc extension of the *Finite State Process* language [Magee, 1999], named *Simple FSP* (SFSP) where transitions are labeled with visible input/output messages, with the following syntax:

!m emitted message *m*

!m$ answer to emitted message *m*
?m received message *m*
?m$ answer to received message *m*

As an example, the specification of the *EmailReader* component depicted in Figure 1.11 is shown in Figure 1.12.

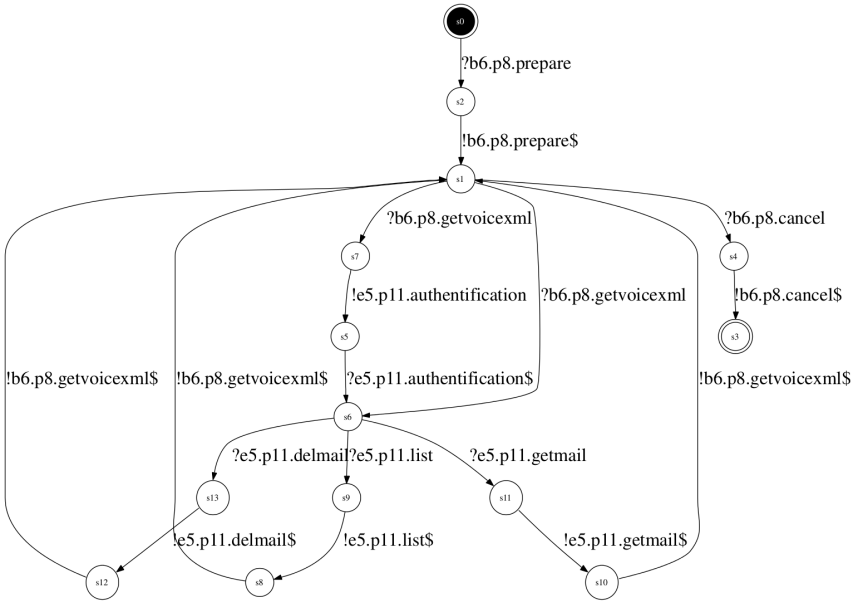


Figure 1.12: Behavioral contract for the *EmailReader* (from [Barais, 2005])

Components behaviors are then formally specified using automata to enable compatibility verifications of component compositions based on their contracts. Further analyses can also be performed regarding *safety* and *liveness* properties of components, since their internal behaviors are specified in an algebraic language.

TranSAT adds an aspect-oriented dimension to SafArchie models [Barais et al., 2005; Barais, 2005]. Alongside to structural architectures, *technical* models focused on specific concerns are specified and weaved into the architecture by model transformations. A TranSAT *canvas* is defined by:

an archi. plan structural pattern that provides an answer to a concern
a joint mask constraints on the possible target model (called *basis* plan)
transfo. rules structural changes to weave the new plan into the basis plan

Figure 1.13 depicts the general process to inject an encryption pattern into a *client-server*-like architecture model.

The basis plan on the top left-hand side of Figure 1.13 specifies a composite component *D* composed by three `components` *A*, *B* and *C*. On the right-hand side, an encryption pattern is defined, composed by the three aforementioned elements, i.e., the joint point mask (*Cm1* and *Cm2* abstract `components`), the encryption architecture plan and a set of transformations linking the elements of the mask and

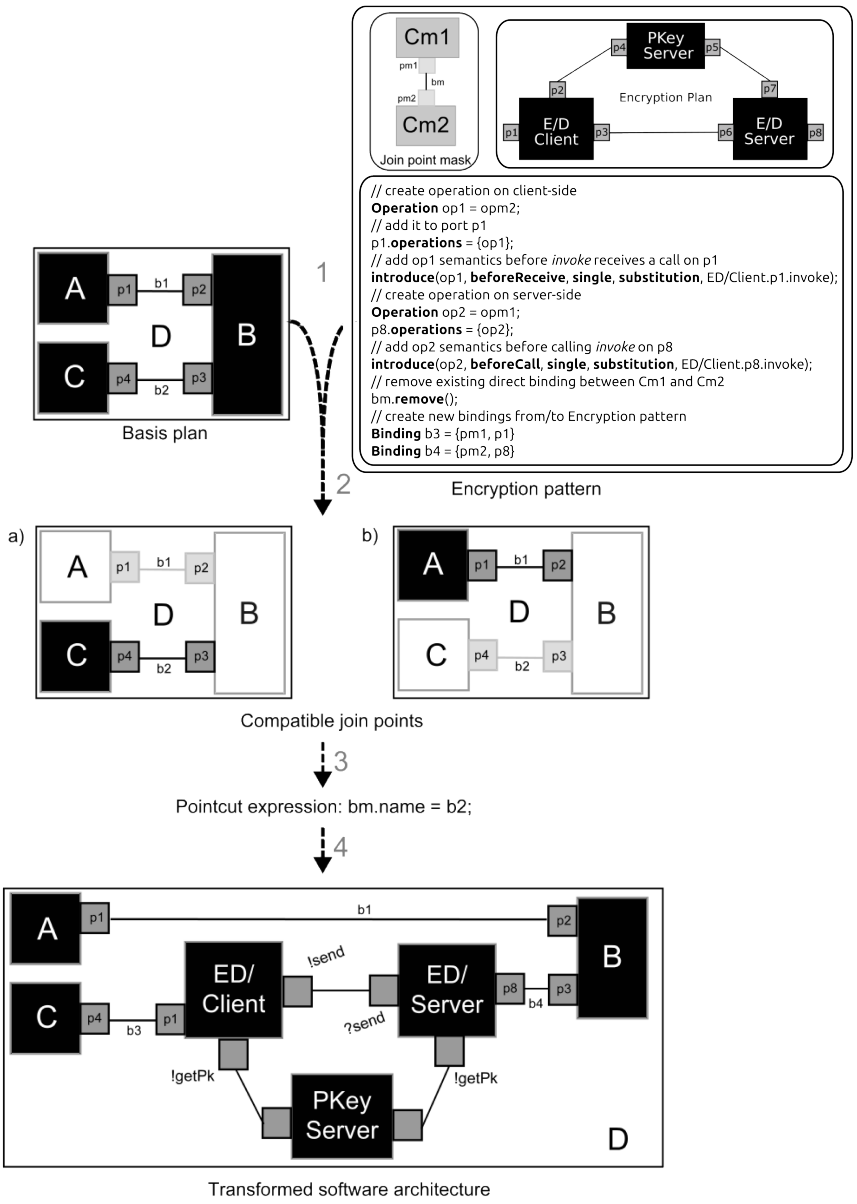


Figure 1.13: Pattern injection example in TranSAT (adapted from [Barais et al., 2005])

the plan together. Two compatible *join points* are identified in the basis plan and one is selected in particular by the *pointcut expression* bm.name = b2. The resulting architecture is depicted in the bottom part of the picture.

On top of these two frameworks, Barais specified an incremental architecture design method around three main *actors*:

domain expert specifies the structural patterns
architect draws the architecture model by weaving patterns
integrator writes the transformation rules and masks

Iteratively, the architect specifies pointcuts and the TranSAT *weaver* executes the selected transformations to produce the new architecture model.

The overall approach is supported by an extension of the ArgoUML tool[15]. The last stable version of ArgoUML dates back from end of 2011 and no new release is planned. Also, only UML 1.4 diagrams are supported and no plan to move to UML 2 is foreseen. SafArchie Studio relies on an unmaintained Prolog framework and is roughly undocumented.

Even if the SafArchie/TranSAT framework is very promising, especially in terms of formal verifications of component compositions, the approach has been abandoned and is not further developed in recent research. Furthermore, the behavioral specification formalism used does not scale very much for the needed abstraction level for software architecture design. The *EmailReader* component behavior presented in Figure 1.12 is barely complicated with only a few external operations. Because of the needed formal semantics for component compositions, `connectors` are synchronous only, which is rather simplistic. Even if the transformation process is extensively detailed, the model versioning question is not addressed and no structural construct is provided to attach documentation to patterns although the TranSAT approach relies on stepwise integration of architectural concerns.

1.2.9 π-ADL and the ArchWARE Development Environment

π-ADL is a formal architecture description language designed to express component-based embedded systems with dynamic concerns [Oquendo, 2004, 2008b]. The language is part of a holistic framework, called ArchWare, that encompasses architecture modeling, analysis, run-time and software development aspects [Morrison et al., 2004; Oquendo et al., 2004].

The ADL is defined on top of the concepts of the π-Calculus [Milner, 1982][16]. Structural and behavioral aspects of systems can be expressed in a *procedural* language with predefined dynamic reconfiguration rules or behavioral changes. Structural models are expressed in terms of `components` connected through `ports` linked by `connections` with `connectors`.

component exposes `ports` and has a behavior
port connects a `component` to its environment
connection simple communication channel between `components`
connector special kind of `component` intended to connect `components`

Hierarchical components can also be expressed in π-ADL, i.e. any architecture model can be encapsulated into a composite `component`. Communication is always done through *value-passing* between `components`. Figure 1.14 illustrates a sample client-server architecture with login facilities.

[15] http://argouml.tigris.org/

[16] π-Calculus is a functional language that focuses on distributed process interactions through dynamic or private communication channels.

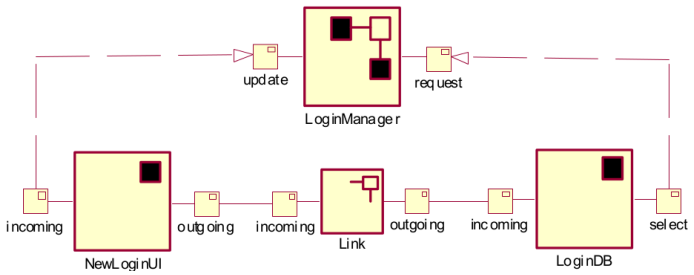


Figure 1.14: Sample client-server architecture in π-ADL (from [Oquendo, 2008a])

Listing 1.6 shows the textual representation of the externally visible ports of the *LoginManager*. Two `ports` are defined. An *update* `port` receives input `connections` with a *Login* value (user defined type). A *request* `port` receives input *connections* with a *UserId* value of type `Any`, i.e whatever value. And an output `connection` sends a `Password` value, also of type `Any`. For the *request* `port`, a `protocol` specifies that a connection is received via the *log* input and sent via the *pwd* output.

```
architecture LoginManager is abstraction() {
  type UserId is Any. type Password is Any.
  type Login is tuple[UserId, Password].
  port update is { connection upd is in(Login) }.
  port request is {
    connection log is in(UserId).
    connection pwd is out(Password)
  } assuming {
    protocol is { ( via log receive any. true*. via pwd send any )* }
  }
}
```

Listing 1.6: *LoginManager* textual specification (from [Oquendo, 2008a])

Behavioral aspects of components are expressed in a *procedural* way. Listing 1.7 illustrates how the *LoginDB* `component` handle login requests.

```
component LoginDB is abstraction() {
  database is location(Set(Login)).
  behaviour is { decrypt is function(p : Password) : Password { unobservable }. }
  choose {
  via incoming::fromLink receive login : Login.
    project login as userId, password.
    database := database' including(tuple(userId, decrypt(password))).
    behaviour()
  or via select::log receive query : Login.
    via select::pwd send (database'
        selecting( lg | project lg as userId, password. userId=query)).
    behaviour()
  }
}
```

Listing 1.7: *LoginDB* textual specification (from [Oquendo, 2008a]

The *LoginDB* component is declared as a storage `location` that saves *Login* values. Its main `behaviour` is a `function` that decrypt incoming passwords and

this `function` will be used in the other clauses afterwards (by calling *behaviour()*). The `choose` clause expresses the variation of behaviors depending on the invoked port. When its *incoming* `port` from the *fromLink* connection receives a *login*, the *database* will add or update the password for this login. When receiving connection on its other `port`, the `component` sends back the *password* for the given *userId*. Other variability and predefined reconfiguration rules can be defined in a similar fashion.

A service-oriented extension of the language as been defined to map *Business Process Modeling Notation* (BPMN) [OMG, 2011a] constructs onto π-ADL [Oquendo, 2008b]. A complete toolset[17] is available with two target virtual machines, one specific to ArchWare and a Microsoft®.NET platform [Qayyum and Oquendo, 2008]. The toolset also provides analyzing, animation and property tracing mechanism.

π-ADL is part of a larger framework for software modeling, analysis and deployment. It has the advantage to rely on a strong algebraic theory for component definition. Properties evaluation is then possible since they are expressed in predicate logic. A UML profile has also been defined to enhance model understandability with a graphical notation.

However, no deployment constraints are considered in π-ADL, but rely on the ArchWare tool. No requirement-related or design rationale can be expressed and, as far as we know, no dedicated mean exists in the toolset. Also, the textual syntax is quite complex to get familiar with. Many keywords are defined and the definition of textual models is somewhat verbose, even if the amount of primitive architectural constructs is rather small. For example, value types must be declared in every component they are used and the communication is done only through value passing. No support is provided for patterns or architectural styles. Finally, the design of an architecture with π-ADL implicitly targets a specific virtual machine under the ArchWare environment. As stated in Section 1.2, we believe an ADL should not restrain the technological possibilities of the underlying technologies, thus we will not consider π-ADL in the discussion in Section 1.2.13, but we thought important to depict the basic concepts of the approach in the present work.

1.2.10 Architecture Analysis & Design Language

Architecture Analysis & Design Language (AADL) is a standard to formally describe embedded and real-time systems [Feller et al., 2006]. AADL offers facilities to model systems in terms of components with many forms of interactions between them, as well as the mapping to computational hardware. AADL has been published as a *Society of Automotive Engineers* (SAE) standard for the first time in 2004 and the last revision was published in 2012 [SAE, 2012]. Many projects have used AADL in the avionics industry [Feiler et al., 2010] and a wide range of research has been published on case studies or methodological aspects for AADL [Wang et al., 2011; Dajsuren et al., 2012; Song et al., 2012]. The core concepts of the original version are depicted in Figure 1.15.

[17] https://www-archware.irisa.fr/software/pi-adl-toolset/

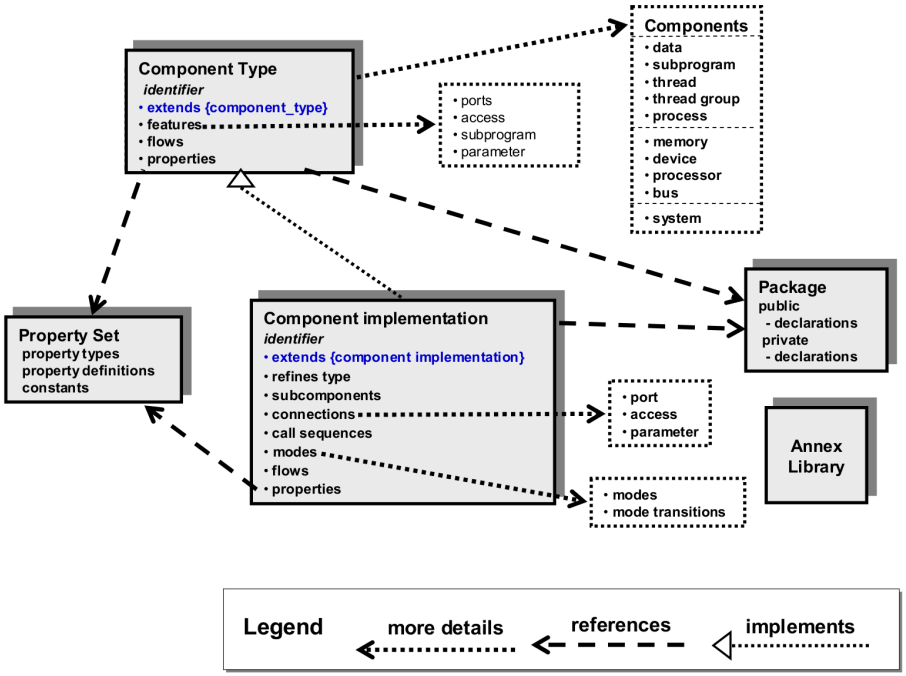


Figure 1.15: Core AADL elements (from [Feller et al., 2006])[18]

Components are defined in term of `types` and `implementations`. A `type` depicts the externally visible interfaces and attributes of a component. The `implementation` specifies the internal structure of a component. As shown on Figure 1.15, there are three categories of components: `application software` (like `threads`, `subprograms` or `data`), `execution platform` (like `processors`, `buses` or `devices`) and `composite` (a hybrid `system`).

A component `type` can be specified by the following *subclauses* that describe more precisely its semantics:

`features`	the interfaces (among predefined interaction types)
`flows`	the abstract information path between components
`properties`	predefined characteristics (dependent on the category)
`extends`	possibility to subclass an existing `type`

Alike, an `implementation` is described by the following *subclauses*:

`subcomponents`	collection of internal components (hierarchal decomposition)
`calls`	`subprogram` calls
`connections`	link between two `features`
`flows`	the implementation of an abstract `flow`
`modes`	alternate configurations (`subcomponents-calls-connections`)

[18] version 2 of the standard mainly introduces syntactical *sugar* and add new types of constructs, like `layered architectures` or `abstract features` (incomplete or *template* component) [Feiler et al., 2012]

properties predefined characteristics (dependent on the category)
extends possibility to subclass an existing `implementation`
refines refine the semantics of its `type` by adding properties

Components can be grouped in `packages` identified by a namespace. AADL is extensible by defining domain specific `property sets` to target components `types`. Also, `annex libraries`, with domain specific notations or formal languages can be developed, but are subject to a formal approval by the SAE AADL committee.

AADL model can be expressed in textual form, in a graphical notation or in XML format. Listing 1.8 presents a sample AADL textual specification.

```
-- a process type definition with an input and an output feature
process control_processing
features
input: in data port sensor_data;
output: out data port command_data;
end control_processing;
-- its empty implementation
process implementation control_processing.speed_control
subcomponents
control_input: thread control_in.input_processing_01;
control_output: thread control_out.output_processing_01;
end control_processing.speed_control;
-- declaration of the thread types and their empty implementations
thread control_in
end control_in;
thread implementation control_in.input_processing_01
end control_in.input_processing_01;
thread control_out
end control_out;
thread implementation control_out.output_processing_01
end control_out.output_processing_01;
-- declaration and implementation of data structure
data sensor_data end sensor_data;
data command_data end command_data;
```

Listing 1.8: Sample AADL textual specification, adapted from [Feller et al., 2006]

In this example, a simple `process` type (containing one input and one output) is specified with its `implementation` containing two `threads`. Many types of components exist in the AADL standard to enable low level descriptions of software systems, with the possibility to refine the semantics with predefined or user-defined properties, like the computation time for a `thread`. Analogously, execution platform components can be specified to represent a physical architecture with `processors`, communications `buses`, physical `memory` and `devices` (interaction points between the system and the environment, like screens, sensors, etc). Once both software and hardware descriptions are complete, software components must be bound to platform components in order to specify a complete `system`. This binding is done via `binding properties` that can be either allowed bindings, actual bindings or available physical resources.

In Listing 1.9, we present the `application` model of a producer-consumer `system` expressed in AADL.

```
package ProdConsSoft -- namespace
public -- visibility
  -- a user-defined data structure
```

```
  data chunk
  end chunk;

  -- the producer program
  subprogram Produce
  features
    output : out parameter chunk;
  end Produce;

  -- the consumer program
  subprogram Consume
  features
    input : in parameter chunk;
  end Consume;

  -- sending thread
  thread Send
  features
    output : out event data port chunk;
  end Send;

  thread implementation Send.Impl
  calls
    prod : { producer : subprogram Produce; };
  connections
  parameter
    producer.output -> output;
  end Send.Impl;

  -- receiver thread
  thread Receive
  features
    input : in event data port chunk;
  end Receive;

  thread implementation Receive.Impl
  calls
    cons : { consumer : subprogram Consume; };
  connections
  parameter
    input -> consumer.input;
  end Receive.Impl;
end ProdConsSoft; -- end of model
```

Listing 1.9: AADL `software application` model for a producer-consumer system

We created two `subprograms`, namely *Produce* and *Consume* that are running as separated `threads`. The *Send* `thread` pushes `events` of type *chunk* and The *Receive* `thread` receives `events` of the same type `chunk`.

Now, we are going to encapsulate the `threads` in `processes` in order to be able to deploy them on hardware nodes. In Listing 1.10, we present the `composite` model gathering `platform` and `system` components.

```
package ProdConsHard
public
  with ProdConsSoft;

  -- the producer process
  process Prod
  features
    outport : out event data port ProdConsSoft::chunk;
  end Prod;

  process implementation Prod.Impl
```

```
12    subcomponents
13      prodprocess : thread ProdConsSoft::Send.Impl;
14    connections
15      port prodprocess.output -> outport;
16    end Prod.Impl;
17
18    -- the consumer process
19    process Cons
20    features
21      inport : in event data port ProdConsSoft::chunk;
22    end Cons;
23
24    process implementation Cons.Impl
25    subcomponents
26      consprocess : thread ProdConsSoft::Receive.Impl;
27    connections
28      port inport -> consprocess.input;
29    end Cons.Impl;
30
31    -- a processor
32    processor the_processor
33    features
34      ETH : requires bus access Ethernet_Bus; -- need access to ethernet bus
35    end the_processor;
36
37    -- a communication bus
38    bus Ethernet_Bus end Ethernet_Bus;
39
40    -- the description of the system
41    system ProdConsSystem end ProdConsSystem;
42
43    system implementation ProdConsSystem.Simple
44    subcomponents
45      prodinstance : process Prod.Impl; -- producer process instance
46      consinstance : process Cons.Impl; -- consumer process instance
47      CPU : processor the_processor; -- shared CPU
48      the_bus : bus Ethernet_Bus; -- shared ethernet bus
49    connections
50      bus access the_bus -> CPU.ETH; -- declare access from ethernet to CPU
51      port prodinstance.outport -> consinstance.inport { -- connect both instances' ports
52        -- actually connected on the same bus
53        Actual_Connection_Binding => (reference (the_bus));
54      };
55    properties
56      -- prod process running on shared CPU
57      Actual_Processor_Binding => (reference (CPU)) applies to prodinstance;
58      -- cons process running on shared CPU
59      Actual_Processor_Binding => (reference (CPU)) applies to consinstance;
60    end ProdConsSystem.Simple;
61  end ProdConsHard; -- end of model
```

Listing 1.10: AADL `composite` model for a producer-consumer system

Two `processes` are specified, respectively for the *producer* and the *consumer* `thread implementations`. The *Prod* `process` declares one `feature`, which is an output `event` of type *chunk* and analogously, the *Cons* `process` declares a type-compatible input `event`. Both `process implementations` expose `ports` typed by the aforementioned `features`. Additionally, a simple processor and a (dummy) communication bus are also defined. Afterwards, a concrete `system` instance is specified with two `processes`, one `processor` and one `bus`. A connection is specified between the `bus` and the `processor` and both `processes` are also connected using their respective `ports` through the `bus`. Finally both `process` instances, namely

prodinstance and *consinstance* are running on the *CPU* `processor`.

A new `annex` is currently being developed to add a requirement modeling language, the *Requirements Definition and Analysis Language* (RDAL) [SAE, 2012]. RDAL provides hierarchical decomposition of requirements (close to the KAOS language [van Lamsweersde et al., 1991]), constraints over requirements, design rationale documentation and a traceability mechanism between requirements and AADL model elements.

Plenty of tools are available on the market to write and analyze AADL models, like *OSATE 2* open source Eclipse plugin[19], to generate application code, like *Ocarina*[20], or refine AADL models like *Ramses*[21]. These tools are supported by a large community and OSATE is a very stable and usable tool developed on top of the Xtext[22] DSL development framework and the Eclipse Graphical Editing Framework[23].

We do not detail here all features offered by the AADL modeling language, like `flows`, `properties`, `modes` and inheritance capabilities. AADL is a very broad modeling language putting a strong relation between software and hardware concerns. Also, the separation between `types` and `implementations` of components, coupled to the `mode` mechanism to define alternative `system` configurations upon given conditions and properties, makes it a very operable modeling language. Furthermore, the *programming language compliance* [SAE, 2006] and *behavioral* [SAE, 2011] annexes places AADL as a good candidate for effective model driven engineering of critical software systems. For example, among other aspects, behavioral execution of `subprograms` or `threads` can be specified by state machines. Transitions are triggered on the receptions of events, by the evaluation of boolean expressions, the combination of both or periodically in case of periodic `threads`. States can be complete so that the execution of the state machine is stopped until further awakening condition occurs.

However, even with limited properties and a very simple instantiation and deployment of a system, the textual representation of an AADL model contains many lines of code and is quite complex to manipulate, like we have seen with our *Producer*/*Consumer* example in Listings 1.9 and 1.10. Also, an AADL model is very close to an *implementable* product, making very tight the difference between *platform independent* and *specific models*. AADL has been designed for embedded and critical systems with analysis possibilities, so it needed a very precise semantics, which drastically augment the amount of modeling constructs and further increases the models complexity. The AADL modeling language is not really suitable to model component-based architecture models at a high abstraction level since even at for application model, a certain level of details is required when operational features need to be expressed.

[19] https://wiki.sei.cmu.edu/aadl/index.php/Osate_2
[20] http://libre.adacore.com/tools/ocarina/
[21] http://penelope.enst.fr/aadl/wiki/Projects#RAMSES
[22] http://www.eclipse.org/Xtext
[23] http://www.eclipse.org/gef/

1.2.11 A Few Words about the MARTE UML Profile

The *Modeling and Analysis of Real-Time and Embedded Systems* (MARTE) UML profile has been designed, as its name suggests, for *Real-Time and Embedded Systems* (RTES) [OMG, 2011e]. MARTE provides, on one hand, constructs for software and hardware aspects of systems with support of non-functional properties. On the other hand, it provides constructs that can be used for analysis purposes, especially for *schedulability* and *performance* properties. MARTE relies on the following models:

Non-func. Properties	notation for non-functional properties specifications
Time	notation for time and clock specifications
Generic Component	component-based view with flow specifications
High-level Application	real-time and behavioral specifications
Detailed Resource	software and hardware platforms properties

Coupled to these notations, the specification defines a textual syntax, namely *Value Specification Language* (VSL) to write expressions in MARTE models. VSL extends the UML `ValueSpecification` and `Datatype` constructs to specify parameters, relationships between these parameters and composite structures.

In many ways, MARTE is close to the *Architecture Analysis and Design Language* we presented in Section 1.2.10, but with a narrower scope, especially for code generation aspects and target platform specifications. It has the advantage to add a precise timing semantics for UML-based components, so model analysis can be performed. In order to represent a full system, MARTE must be combined to SysML `block` constructs [OMG, 2012d]. The UML profile is also relatively complex to manipulate and provides many additional concepts to UML models, which further increases its complexity. MARTE plugins are available for, among other, MagicDraw, a commercial modeling tool for many languages, including UML [24] and Papyrus, an open source Eclipse plugin with support for almost the overall UML models and many profiles [25].

1.2.12 The Open Group ArchiMate® Modeling Language

Enterprise Architecture (EA) is defined as a broader discipline that integrates software architecture as one of the aspects of an enterprise. EA focuses on guiding businesses with a holistic approach, usually by integrating business goals, strategical aspects, people and technologies. From a set of recognized EA definitions from the industry and the academic worlds, Dankova stated that « *enterprise architecture represents (an approach to developing) a general conceptual plan, which describes the structure of the enterprise with its separate components and links between them; it defines the principles and rules for the design and operation of the organization structure, the processes and information systems in the enterprise, and it synchronizes information technologies in the enterprise with its business goals and processes.* » [Dankova, 2009]

EA modeling frameworks usually provide mechanisms to support business, data, application and technological architectures [Lapkin et al., 2008; Open Group, 2011;

[24] http://www.nomagic.com/
[25] http://www.eclipse.org/papyrus/

Mentz et al., 2012]. Among architectural modeling languages, the ArchiMate language is quite popular [Malavolta et al., 2013] and offers interesting features for software architecture modeling with requirements traceability [Open Group, 2013]. The ArchiMate is based on a layered representation of an enterprise: *Business*, *Application* and *Technology*. The language core concepts are expressed in terms of *active* elements that perform some *behaviors* on *passive* objects.

The *Business* layer represents the `processes` executed by `actors`. `Actors` may have `roles` that defines their responsibility to perform some behaviors. In case many `roles` are responsible for a behavior, a `collaboration` can be specified to link multiple `roles` to one or more behaviors. `Actors` are placed into `location` that can be either geographical or conceptual. Business `services` are exposed through `interfaces` that represent the point of access from the outside environment. Business behaviors are `processes`, `functions`, `interactions`, `events` and `services`. `Processes` are ordered activities that produce `services`. A `process` can be part of a `service` that fulfills someone's need. Services and processes can be grouped under user-defined criteria in a business `function`. `Events` are occurrences that triggers or influences behaviors. `Interactions` are collaborative `processes`. The standard defines also a set of passive concepts at the *business* layer. An `object` is anything relevant from a business perspective, its physical appearance or format being a `representation` and their contextualized semantics are expressed as `meanings`. A `product` is a collection of correlated services offered to a customer. The relative benefit of a `service` or a `product` is a `value`. A `contract` is the agreement associated with a `product`.

Figure 1.16 depicts the `process` flow of the *Take out insurance* `service` that is triggered by an insurance request `event` and assigned to the *Insurance agent*. The service can be accessed through two `interfaces`, a *Web* page or the *Phone* provided by the *Insurance agent* `role`. The insurance can be rejected, depicted by an `or-junction` that generates a *Request rejected* `event`. If the insurance request is accepted, a *policy* `object` is created and its *paper* `representation` is sent out to the client.

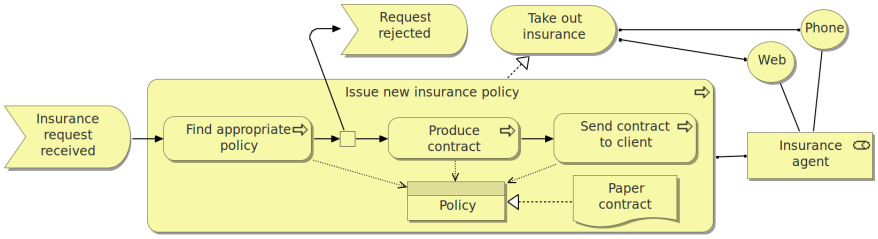


Figure 1.16: Insurance request Business `process`[26]

The *Application* layer is a software-oriented support for the *Business* layer. Close to UML 2 component diagrams, `components` support `functions` exposed as `ser-`

[26]All models presented in this section are inspired from the *ArchiSurance case study* [Open Group, 2013]

vices on interfaces. Internal interactions may be defined into components for application collaborations. *Application* layer behavioral constructs share the same names as the ones at the *Business* layer and are in some ways the translation of business behaviors into software processes. The only application *passive* structure is the data object representing a computable object. Figure 1.17 illustrates an *Application* model. A *Financial Application* component contains two functions, each of them with two services. The invoice-related services access to an *Invoice* data object. An interaction is also described between the *Financial* and *Client* applications to retrieve the *Client* details.

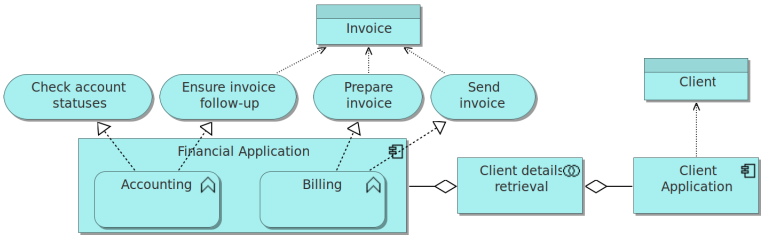


Figure 1.17: Financial application component

The *Technology* layer mixes underlying software technologies, such as operating systems or application servers, and hardware infrastructure. Closely to UML 2 deployment diagrams, nodes and devices can host artifacts. Nodes are bound via communication paths and devices through networks. System software are UML 2 ExecutionEnvironment. But The ArchiMate language introduces also interfaces at the infrastructure level to model logical or physical gateways for other nodes or application components. Figure 1.18 represents a sample infrastructure view with on the left-hand side the *Main Office* location that contains the *Financial* and *Client* application components deployed respectively as a *SAP-FI/CO* artifact on a an *SAP Server* device and as a *CRM.jar* file deployed on an *EJB Container* system software. A *DBMS* offering *Data Access* functions is accessible via a *JDBC* infrastructure interface to the *Client Application*. All these nodes are connected to each other in a *LAN* network. The *Take out insurance* process is accessible on *Client Browser* application interface deployed on a *Client Personal Computer* node.

User-defined views and viewpoints can be specified to abstract part of architecture models depending on the model users, as defined by the ISO/IEC/IEEE standard on software architecture description [ISO/IEC/IEEE, 2011]. A viewpoint defines the intended audience, i.e. stakeholders, the purpose, like communication, design or decision, the abstraction level and the available modeling elements. A set of standard viewpoints as well as a methodological framework to select among these viewpoints is also introduced in the specification document.

Similarly to the UML profiling mechanism, *extensions* can be added to the ArchiMate language. At current stage of development, two standard extensions have been defined, one for *motivational* aspects and one for the *implementation and migration*

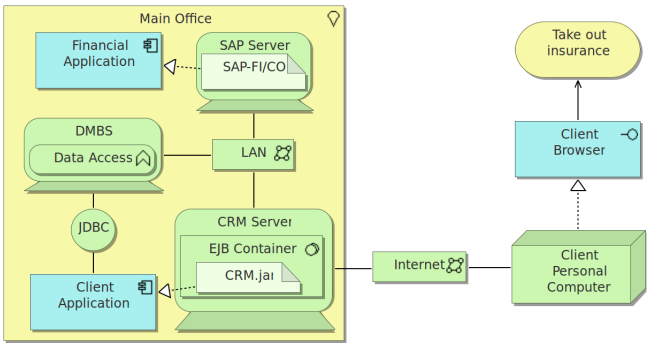


Figure 1.18: Sample infrastructure view

of systems. The latest has been developed to stick to the *Architecture Development Method* (ADM) as specified by the TOGAF® framework [Open Group, 2011][27].

The *Motivation extension* is aimed to represent, among others, analysis `assessments`, business `goals`, system `requirements` or `stakeholders`. Figure 1.19 illustrates how from a *legal obligation to be insured* `assessment` for a *Customer*, a *Contract insurance* `goal` is refined in a *Personalized* and an *Automated policy definition* `requirements`. They both are offered by the *Take out insurance* `service` via specific `interfaces` and implemented by an application `component`.

In our opinion, the holistic approach advocated by Enterprise Architecture modeling approaches have significant assets that could be integrated into software architecture descriptions. Upon its relative complexity induced by the high number of modeling element, The ArchiMate language offers interesting features, especially with the *Motivation* extension that relates requirements to actors or components. The layered architecture representation, coupled to adequately used viewpoints, is also an effective mean of communications for non IT specialists. The possibility to model physical gateways with `interfaces` provides also a more flexible and accurate way of modeling concrete bindings between `nodes`. Model versioning is somehow addressed by the *implementation and migration* extension within the `plateau` construct where an overall architecture is stored at specific points in the ADM method.

However, since it mainly focuses on business-oriented features, application `interfaces` are poorly defined and logical connections between processes through these `interactions` cannot be formally expressed. `Services`, at business as well as application layers, and `data objects`, that partially define an `interface` semantics, are also specified in an informal manner. Besides, the nature of the communication in a `collaboration` or in an `interaction` is also expressed in a fuzzy way where the exact content of the exchange information, data, or anything whatsoever

[27] TOGAF is outside the scope of the present dissertation. In brief, an ADM cycle is composed by eight repeating tasks (also divided into steps): architecture vision, business architecture, information systems architectures, opportunities and solutions, migration planning, implementation governance and architecture change management. The first cycle is usually preceded by a preliminary phase.

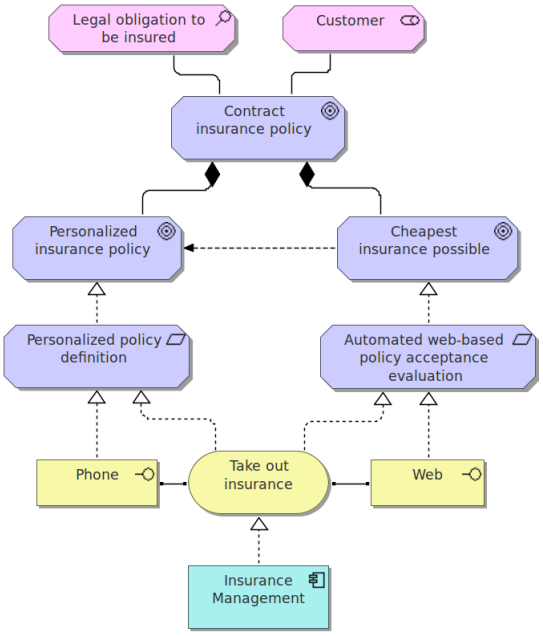


Figure 1.19: Motivation extension sample view

has no semantics attached. Furthermore, rationale or alternative design are not documented and no modeling constructs are offered to keep track of such piece of information. Last, the behavioral specification mechanism are rather informal so that verification processes, even manual, are quite complex to perform.

1.2.13 What Are They Missing ?

We will now review the aforementioned architectural modeling approaches regarding the four key aspects we introduced in Section 1.2.2.

Design Facilities

They all provide model elements to represent an architectural definition, except MARTE that must be combined to SysML, D&C that focuses on deployment management and ArchiMate that offers few semantics on components interfaces. xADL provides only syntactical blocks with no semantics, so they need to be refined by the modelers. However, except for UML 2 and SysML which allow the definition of behavioral specifications on connectors, the concrete semantics of the binding between abstractly defined components or interconnected computation nodes is either trivial, not clear, or even sometimes unspecifiable. AADL offers many interesting features to refine the semantics of any model element by separating the *type* and *instance* levels, but the amount of primitive construct is so large that a complete

architecture for a simple system, such as the *producer/consumer* (Listing 1.10 in Section 1.2.10) is already pretty complex.

Except ACME and D&C, all investigated languages offers multiple viewpoints for architectural model. Coupled to structural representations, architecture models can be extended at least by behavioral specifications at different levels of detail or formality. For example, ArchiMate mainly focuses on business processes, UML/SysML lets users free to define their own level of details and xADL is extended with state machine features. At the opposite, SafArchie, requires modelers to specify the component behavior with a labeled transition system based on a strong algebraic theory. Regarding platform or deployment viewpoint, all languages excluding ACME, roughly provide the same constructs, but with different level of expressiveness[28].

Also, model refinement, enrichment and *versioning* are scarcely addressed by these modeling approaches. Usually, modelers are in charge of maintaining the links between different versions or views of a model. SysML enables to represent systems at different abstraction levels with nested BDD-IBD. ACME offers a somewhat similar mechanism by distinguishing between a component and its possibly recursive internal representation. Modifications applied to models, especially when developing in an iterative manner, cannot be easily or formally retrieved so that exploration and documentation of design alternatives need extra work to be efficiently maintained. AADL and MARTE share a common view of system `mode` for dynamic reconfiguration under certain circumstances, but do not specifically support iterative enrichment and versioning of models.

Finally, except ACME and UML 2 components, the definition and reuse of architectural patterns or styles is even complex, or not addressed by the aforementioned approaches.

Mean of Communication

Another prior asset for modeling languages resides in their communication faculty. Models are not designed for practitioners only, but also for stakeholders. An important trade-off must be found between formal and generic notations. More formal and domain-specific approaches empower a higher expressiveness and facilitate model analysis, but lower the understandability by non-practitioners. At the opposite, lazy or fuzzy construct semantics usually avoid analysis possibilities. In the SafArchie 3-layer representation, component and port semantics must be expressed by algebraic automata which may be too difficult to understand, manipulate and maintain. AADL, in its second annex, share a close approach by providing a formal syntax to define execution semantics with extended state machines. ACME has a completely opposite approach where constructs have a generic semantics than can be refined by properties[29]. In the middle of these antagonist approaches, UML-based formalisms and ArchiMate use constraints or semi-formal behavioral models

[28] since MARTE is a UML profile, it may reuse UML deployment diagrams.

[29] Note that its FSP extension allow to formally define component behavior in a similar fashion as AADL behavior annex, but since the approach is rather undocumented, we do not consider it to be part of the language.

to provide human-readable execution semantics that can be used as a basis of communication of verification of models. xADL relies once more on the user-defined extensions to properly specify the semantics of the models.

Furthermore, the number of primitive constructs, which influences the language complexity, should be considered. With complex meta-models, the effort to understand and maintain models can become more complex, as observed with UML 2 [Solberg et al., 2005]. Again, an important trade-off must be considered between a high expressiveness and an explosion of primitive constructs to combine in order to draw a model [Sen et al., 2009]. AADL and MARTE are made of many primitive constructs, so that, even for toy samples, architecture models contains many line of codes, making them quite complex to understand. But with ACME models, constructs must be refined by non-standard properties, so the understandability relies mainly on the level of documentation of the semantics of these properties.

Model Analysis

Regarding model analysis, some languages propose interesting features. SafArchie integrates a framework for compositional verification based on algebraic automata. MARTE has been conceived to support model verification regarding non functional properties, especially for schedulability and performance. SysML also support some kind of analysis via `constraint blocks`. AADL models are refined by standard properties and can be extended by behavioral aspects for analysis purposes[30]. ArchiMate provides analysis support via dedicated types of diagrams, but only for business-related concerns. ACME components semantics can be refined by FSP definitions supported by an LTS analyzer, but the concrete way to use this extension is not documented even if the FSP formalism and the analyzer tool are actually well documented. xADL proposes basic structural features for model analysis, especially on model consistency checking.

As we can see, analysis is usually performed by the mean of dedicated properties expressed in a formal manner or by transition systems with different level of abstractions. ACME and AADL provide the more flexible features since both mechanisms can be used and state machines must be expressed in a formal manner, which suit particularly to effective model analysis.

Many recent approaches for behavior specifications based on the *Foundational UML* (fUML) initiative [OMG, 2013c] and the *Action Language for fUML* (Alf) [OMG, 2013a] are emerging [Lai and Carpenter, 2012; Mayerhofer et al., 2013]. fUML defines a subset of UML concepts with a precise execution semantics. The Alf language is a textual Java-like syntax to describe and manipulate fUML models. The combination of fUML and Alf enables simulation and verification possibilities. These approaches make fuzzier the line between modeling and programming, but does it really matter? We believe such behavioral specification mechanisms better suits the needed abstraction level for software architectures.

[30] Further domain specific annexes can also be written, but are subject to approbation by the *Society of Automotive Engineers* board.

Requirement Linking

A needed feature for today's modeling languages is the linkage from requirements to software artifacts in order to face, even partially, *architectural drift* and *erosion* [Perry and Wolf, 1992]. Architectural drift may be seen as the gradual incoherence of the *concrete* architecture in terms of its deployed artifacts regarding to the modeled architecture. Such a drift is due to the evolution of the system, but it does not break any constraint in the model. At the opposite, the erosion is a similar phenomenon that violates the architectural model. Exposed to these types of degradations, a system architecture may become too *obscure* and hermetic to changes.

Even in UML component diagrams where there could have been a link to *Use Cases* for example, or in TranSAT with the iterative injection of patterns, this linkage is not explicitly kept so that there is a need to define and maintain extra models or documentation for that purpose. SysML and ArchiMate are the only explored languages that are providing this linkage between component-like artifacts and addressed requirements. They also provide refinement traceability between requirements. In SysML, design rationale can also be added to any modeling elements, including *associations*. ArchiMate is completed with a *Motivation Extension*, but uses rationale in the opposite way. `Assessments` and `Drivers` are used to derive `Requirements`, then the `Requirements` are linked to `Application Components` or to `Actors`. However, alternative solutions or refinements of requirements as well as impacts between requirements cannot be expressed. Within SysML, an extensible `trace` dependency can be defined between requirements but it is up to the modeler to specify its semantics. This mechanism offers the possibility to create *ad-hoc* traceability associations like, for examples, inter-dependencies between requirements or mutual exclusions.

Summary Table

Table 1.1 summarizes our appreciations of the explored languages regarding our four key aspects. Ratings are expressed with qualitative values compiled from our aforementioned remarks:

- – major lacks concerning this aspect
- o most of the aspect is taken into account
- \+ the aspect is appropriately taken into account

KEY	UML2	D&C	SYSML	ACME	XADL	SAFARC	π-ADL	AADL	MARTE	ARCHIM
DESIGN	+	–	+	o	o	+	o	+	o	–
COMM.	+	–	+	o	–	o	–	–	–	–
ANALY.	o	o	o	o	o	+	+	o	+	+
LINK	–	–	+	–	–	–	–	o	–	+

Table 1.1: Summary of appreciations regarding the key aspects

1.3 Design Rationale and Requirement Traceability

The value of appropriate documentation for software design and maintenance has been widely stressed by the scientific community [Royce, 1970; Parnas and Clements, 1986; Curtis et al., 1988; Watkins and Neal, 1994; Clements et al., 2002; Avgeriou et al., 2007] and in the industry [Tang et al., 2006; Ali Babar et al., 2006, 2007; Malavolta et al., 2013]. It is even more important as design rationale and knowledge is often diluted into the overall design and tends to *evaporate* with system evolutions, because the knowledge mainly remains in the head of some key experts or architects [Rus and Lindvall, 2002; Zdun, 2009]. Recording techniques for design rationale has been extensively studied for years and many tools and models have been proposed with few success in practice [Jarczyk et al., 1992; Shum, 1996; Tang et al., 2006]. These early methods were mainly text-based with a lack of reasoning and analysis capabilities [Ali Babar et al., 2006]. Furthermore, the perceived *return on investment* of such techniques is usually very low since the needed additional work does not produce its benefits right away.

In the software architecture community, Bosh and Jansen postulated that an architecture can be seen as the set of decisions that produced the model, with the explored and discarded alternatives accompanied by the reasons sustaining this model [Bosch, 2004; Jansen and Bosch, 2005]. From the suggestion made by Tyree and Ackerman to explicitly record and model architectural design decisions [Tyree and Akerman, 2005], Kruchten *et al.* identified four types of decisions and introduced a first classification for design decisions and their rationale [Krutchen, 2004; Kruchten et al., 2006]. A set of *Architecture Knowledge* (AK) frameworks and tools have then emerged in the past decade to capture the rationale behind the production and evolution of architecture models [Bjørnson and Dingsøyr, 2008; Farenhorst and de Boer, 2009]. In the present section, we will review notable and recognized methods, models, frameworks and tools in the field of AK. We do not provide a systematic literature study[31], but we tend to highlight recurrent needed traceability features for design decisions and rationale that should be incorporated into a component-based modeling framework.

1.3.1 Architecture Tradeoff Analysis Method[SM]

Kazman *et al.* introduced the *Architecture Trade-Off Analysis Method* (ATAM) to analyze software architecture regarding its expected quality attributes [Kazman et al., 2000]. ATAM is a risk-oriented method intended to help architects to make the correct *trade-offs* at early design stages. ATAM tends to make explicit the relations between the model quality attributes, i.e. the software *ilities*, and their responsible structural elements and properties. This is thus a decision *reasoning* and knowledge *recording* technique about architectural design and decisions based on the identification of:

risks dangling or unevaluated decisions

[31] Such reviews are available in an article written by Bjørnson and Dingsøyr [Bjørnson and Dingsøyr, 2008] and in a technical report written by Biehl [Biehl, 2010]

sensitivity points architectural elements liable to quality attributes
tradeoffs elements being multiple *sensitivity points*

The method is organized around nine steps, some of them may be repeated twice, and is supported by predefined document templates and organization agenda. In a nutshell, the steps are grouped in *presentation*, *investigation and analysis*, *testing* and *reporting* tasks. The *presentation* focuses on presenting the ATAM, the business driver and the proposed architecture. In the *investigation and analysis*, architects first identify architectural approaches. Then, they define and prioritize quality attributes, and define scenarios that conform to these quality attributes in so-called *utility trees*. Finally, they analyze the approaches regarding the high-priority attributes. The *testing* consists in the elicitation of a larger set of scenarios with all stakeholders, not only the architects present during the previous phase and they all analyze these scenarios and possibly produce new *risks-sensitivity points-tradeoffs*. Last, the results of the ATAM process are presented to the stakeholders in the *reporting* task.

The ATAM method also defines a *characterization framework* illustrated for *performance*, *modifiability* and *availability* quality attributes. They identified three categories:

external stimuli causes behind architectural changes
architectural decision structural changes induced by the *stimuli*
responses measurable effects of an *architectural decisions*

This characterization is not meant to be fully exhaustive, but the goal of the method is to propose a framework to reason about quality attributes. It focuses then on the impact analysis and elicitations of architectural decisions regarding quality attributes, exclusively in terms of structural *responses* to architectural changes. It has been developed to make early analysis and enforce more accurate documentation to evaluate the risks induced by architectural changes. The proposed documentation does not consider other type of rationale regarding the design, such as alternative solutions or technological/hardware constraints, for example.

1.3.2 4 + 1 View Model of Architecture

Kruchten introduced a composite representation of software architecture based on the following four views [Kruchten, 1995]:

logical platform independent structural features of an architecture
process the communication, timing and other non functional aspects
development platform dependent and detailed system description
physical the mapping to hardware with physical constraints

A *scenarios* view compose the glue between all these representations gathering a subset of software requirements. It does not provide additional information, but is an interesting mean to illustrate and validate the design on paper, as well as discovering architectural aspects in early design phases.

From the definition given by Perry and Wolf, as presented in Section 1.2.1, all views are augmented by an explicit recording of the rationale and constraints. Architectural elements are also bound to some of the requirements. This method has

been fine tuned and used at Rationale Software with some success, according to the author. As far as we know, this is the first report of such an explicit recording in a software development method. However, no concrete details on the form or on the nature of these recordings are provided.

1.3.3 Process-based Knowledge Management Environment

Ali Babar *et al.* proposed a methodological framework to record tacit knowledge in architecture-based system development [Ali Babar et al., 2005a]. The idea relies on a dedicated process decomposed into four tasks: *planning*, *capture*, *organization & validation*, and *storage*. During the *planning* phase, the nature and sources of knowledge are identified and an appropriate recording technique is selected. The knowledge *capture* is organized in two ways: through the implication of the development team with explicit tool-supported recordings, or by a dedicated knowledge engineer in charge of the acquisition and recording either during the project or from a retrospective exploration of patterns. In order to appropriately extract architectural knowledge from patterns, they must have been documented at least as *reusable patterns* with, among other information, the addressed problem space and the intended consequences [Buschmann et al., 1996]. Afterwards, the knowledge must be organized and validated, but, unfortunately, the authors do not give any details on this third phase and do not point to any other method or technique. Last, this validated information must be stored in a repository and made available to others.

A web-based tool has been developed on top of the *Hipergate* collaborative and open-source platform[32] to support the overall framework [Ali Babar et al., 2005b; Ali Babar and Gorton, 2007]. It has been live-tested in an Australian Defense project for an aircraft system [Ali Babar et al., 2008]. Upon some advantages of recording and using architectural knowledge in an industrial environment, the experiment notably highlighted the need for variability and design alternatives management as well as an appropriate pattern-based documentation.

1.3.4 Architectural Design Decision, a First Classification

Based on the observations and suggestions by Bosch [Bosch, 2004] and Tyree and Ackerman [Tyree and Akerman, 2005] to raise design decisions as *first-class* modeling artifacts in the software development life cycle, Kruchten *et al.* proposed an ontology of design decisions [Krutchen, 2004; Kruchten et al., 2006]. The authors identified four types of decisions from *implicit and undocumented* to *explicit and documented*. They defined a classification made up four classes, their attributes and relationships. Design decisions are either:

ontocrises existence decisions, *i.e.* an element is present in the design
anticrises opposite of *ontocrises* stating that an element will not exist
diacrises property decisions that express a quality of (part of) the system
pericrises organizational decisions influencing the development life-cycle

[32] Hipergate is a Content Resource Management tool, see http://www.hipergate.com

Any design decision is refined by attributes to capture as much information as possible, with among others, a text based description, its rationale, a scope, its state (among predefined values) and the history regarding the authors, dates and successive changes made to the decision.

Also, numerous relations can exists between two decisions or between a decision and another artifact, like an element present in another model or a concrete system. Twelve types of relations where defined to this end:

constrains	a lower-order decision cannot live without a higher-order one
forbids	a decision excludes another one
enables	a decision makes available, but does not implies another one
subsumes	a decision is more general, i.e, surround another one
conflicts with	mutual exclusion of decisions
overrides	exception to another decision that has a wider applicability
comprises	decomposition of a higher-order decision
is bound to	both decisions must be in the same state
is an alternative to	fully substitutable decisions
is related to	any other type of decision not listed above
traces to	relation between a decision and an artifact (code or model)
does not comply	a concrete artifact prevents from taking a decision

This ontology stated the basis for much research and many tools focusing on architectural knowledge management. We will present some of them in the following sections.

1.3.5 Architecture Rationale and Element Linkage

Tang *et al.* introduced the *Architecture Rationale and Element Linkage* (AREL) model to capture and trace design rationale[Tang et al., 2007]. The AREL model traces the reasoning process and the history behind an architecture model. The authors made a distinction between *motivational reasons* and *design rationale*. The *rationale* is produced by the decision and justifies it with details such as the alternative options and tradeoffs. A motivational reason is an input to a decision and is either:

causality	incentive for a design decision
goal	objective to be achieved
influence	decision that can constraint another one
factuality	factual piece of information or assumption

The AREL model has been implemented as a UML profile for adoption and compliance reasons, since UML models are relatively popular in the industry. Architectural rationale are concretely linked to any architectural element with a single type of traceability link. One rationale can be linked to many elements, but only one rationale can be bound to an element. AREL considers architectural elements from four different perspectives, i.e. viewpoints in the sense of the ISO/IEC/IEEE standard [ISO/IEC/IEEE, 2011], as defined in the TOGAF [Open Group, 2011] methodological framework: *business*, *data*, *application* and *technology*. AREL models are also extended to capture versions of architectural elements in order to retain information over system evolution.

AREL has been implemented as an *Enterprise Architect* extension[33]. It requires to draw extra models and the evolution traceability mechanism has not been fully implemented because of compiler compatibility reasons. The approach has been retrospectively tested on a partial case study to retrieve and encode architectural decisions, but the reasoning capabilities of AREL and whether it helps in designing software systems have not been evaluated.

1.3.6 The Core Model and the GRIFFIN Collaborative Community

From an initial conceptual model for *Architectural Knowledge* (AK), de Boer *et al.* conducted an experiment in four different organizations with distinct perspectives on what they actually consider as part of the AK [de Boer et al., 2006, 2007]. The main concerns of these companies were either on *effective knowledge sharing*, *architectural rules compliance*, *AK recovery* and *AK traceability*. The initial model was highly compliant to the IEEE-1471 standard [IEEE, 2000] but the experiment revealed some discrepancies with the organizations' views. Instead of growing their models with new entities, the authors created a minimalistic *core* model by gathering closely-related concepts. The resulting model does not intimately relate to the aforementioned IEEE standard but leaves more freedom on the actual architecture description formalism. The authors also validated their model to other AK formalisms present in the literature, such as the ontology introduced by Kruchten that we presented in Section 1.3.4.

The core model is articulated around twelve concepts:

`stakeholder`	any party that has an interest in the system
`concern`	actual interest of a `stakeholder`
`artifact`	any tangible and meaningful piece of information
`language`	representation formalism or notation of `artifacts`
`archi. design`	combination of `artifacts`
`decision topic`	`concern` that must be addressed by a `decision`
`alternative`	possible *implementation* option
`ranking`	classification value attached to an `alternative`
`decision`	selected `alternative`
`design decision`	`decision` that has an impact on the architecture
`activity`	action that produces or uses `artifacts`
`role`	`stakeholder`'s *hat* in `activities`

The same authors also propose a sort of *open-source* virtual community upon their core model serving as a reference model for AK management [Lago et al., 2010]. They argue for a *hybrid* strategy that mixes *codified* and *personalized* knowledge management [Hansen et al., 1999]. Codification consists in carefully recording knowledge in dedicated storage, sometimes made available in a standardized manner. Personalization is the opposite method where knowledge stays in people's mind and is shared via direct human contacts. Extreme codification strategies induce a significant effort for documentation activities where highly personalized ones live

[33] Enterprise Architect is a commercial tool for the overall software development life-cycle and it includes UML modeling facilities. See http://www.sparxsystems.com

under the threat of loosing knowledge with employee turnover. The authors envision a social network where professionals may publish (part of) their own AK on *sharing*, *discovery*, *traceability* and *requirement compliance* techniques. They argue that such a community, supported by the industry, will provide reusable practices and solutions to common AK management problems.

1.3.7 Viewpoint-Based Documentation Framework

Based on the conceptual model and definitions defined in the ISO/IEC/IEEE 42010 standard on software architecture [ISO/IEC/IEEE, 2011], van Heesch *et al.* proposed a documentation framework composed by dedicated viewpoints for architecture decisions [van Heesch et al., 2012]. All viewpoints are compliant with the definition of an *architecture framework* as stated in the standard, i.e. *« conventions, principles and practices for the description of architectures established within a specific domain of application and/or community of stakeholders »*. The conceptual meta-model of an architecture framework is depicted in Figure 1.20.

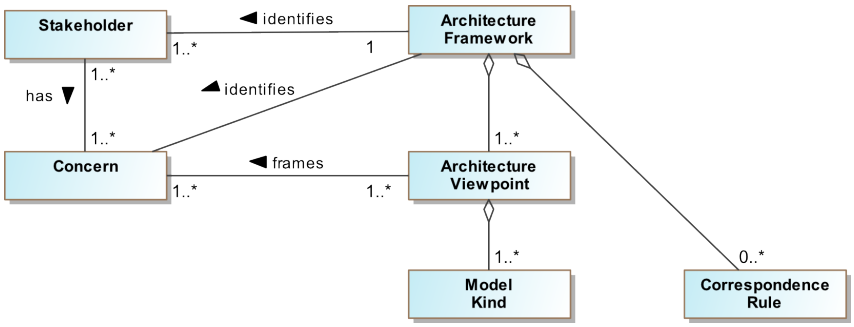


Figure 1.20: Architecture framework meta-model (from [ISO/IEC/IEEE, 2011])

First, the *Decision Detail* viewpoint is a text-based description and concentrates on the design decision with, among others, the explanation of the addressed problem, the decision with its possible alternatives, the related decisions and requirements and the history with the authors of earlier revisions. Since the expected level of details is rather high, the authors observed during pilot studies that only important decisions were recorded using this viewpoint.

Second, the *Decision Relationship* viewpoint focuses on a *snapshot* of the decisions with the relations between them and their statuses. The framework does not define the precise semantics of the relationship so that domain or organization specific types of relations may be defined.

The third viewpoint addresses the responsibilities of the stakeholders involved in the decision-making process. A *Decision Stakeholder Involvement* view aims at modeling personalized knowledge by connecting stakeholders to decisions. It also capture a temporal view of the development process since this viewpoint must be defined for a specific development iteration.

Last, the temporal evolution of decisions is depicted from a *Chronological* viewpoint that represents the decisions with their statuses along the architecture iterations. This viewpoint is particularly useful to understand the history of an architecture development and the decisions-making chronology of the overall process.

Except for the *Decision Stakeholder Involvement* viewpoint, the approach was validated in an industrial case study. The authors observed a positive effect on the communication between stakeholders concerning the *technical* architecture without requiring an unacceptable overwork for documentation.

In a comparative case study, the authors tested their framework on software engineering students working on industrial projects [van Heesch et al., 2013]. As a result of the study, they observed that their framework induced a more systematic exploration of design alternatives, but did not significantly helped at managing system complexity.

1.3.8 Issue-Based Information Systems

Issue-Based Information Systems (IBIS) have been introduced by Kunz and Rittel to structure issues and their solutions in collaborative problem-solving activities [Kunz and Rittel, 1970]. The main asset of such systems is to keep track of the argumentation and decision processes related to a particular problem or topic. Many tools have been developed based on the IBIS concepts in the past years [Shum et al., 2006]. In the present section, we give an overview of two of them.

A rationale capture mechanism integrated into the Eclipse environment has been introduced in the *Software Engineering Using RATionale* (SEURAT) system [Burge, 2005; Burge and Brown, 2008]. Even if the main goal of the approach is on rationale capture for the complete system life-cycle, the authors initially focused on maintenance activities and defined an argumentation-based language called RATSpeak, inspired by the *Decision Representation Language* [Lee, 1989, 1991]. The argumentation starts from a `decision problem` that can be refined in sub-decisions, solved by `alternative solutions` with `arguments`. `Questions` can be made during the decision process and express the need for additional information before further argumentation. System `requirements` are derived from `arguments`. `Arguments` can also be justified by `claims` or `assumptions`, the latter being *system-specific* and unreliable `claims` that may be dropped at some time. `Claims` must be linked into a predefined `argument ontology`, like *Development Cost* or *Portability*, for example. The approach has been tested in a comparative experiment on maintenance tasks of a Java application. It revealed to be partially useful and time-saving, especially for developers with few or advanced expertise with Java coding.

The *Design Rationale Editor* (DRed), primarily developed at Cambridge University, has been successfully integrated in the standard toolset at Rolls-Royce [Bracewell et al., 2009]. DRed has been initially designed to *diagnose problems, design solutions, complete a standard checklist* and *communicate the design and its rationale*. DRed models are composed by ten primitive elements, among others, by `issues`, `requirements`, `answers` or `arguments`. Any of these model elements has a status attached, for example an `issue` can be `opened`, `resolved`, `insoluble` or `rejected`.

External documents can be linked into a model as well as system `blocks` of the final product. Directed relations may be defined between any type of elements and depicts how the origin influences the destination (`neutral`, `harmful` or `useful`). According to the authors, the tool is an helpful support to structure design thinking, capture design rationale and had the side effect to reduce the need for extra documentation in external reports.

1.3.9 A Few Words about the OMG Decision Model and Notation

The Object Management Group is currently working on a standard for a *Decision Model and Notation* (DMN) [OMG, 2014a]. The main objective of DMN is bridging the gap between business process modeling, such as BPMN [OMG, 2011a] and decision logic, like OMG *Production Rule Representation* (PRR) [OMG, 2009][34]. It is intended to capture details about human and automated decisions in either natural language or in a decision logic formalism, into a *Decision Requirements Graph* (DRG), possibly split into *Decision Requirements diagrams* (DRG). These models can then be used to automate decision-making processes. It also aims at modeling business knowledge with the related business-level decisions. The standard is still under review and targets a *business* audience with interest in automated decision logic, rather than a software engineering community.

1.3.10 Goal-Oriented Requirement Modeling

In the Requirements Engineering field, a quite large community has grown around goal-oriented modeling for early requirements engineering [Mylopoulos et al., 1999; van Lamsweerde, 2001]. Those approaches concentrate on organizational concerns that may lead to software requirements. In a sense, their focus is the application domain, where the late requirements and other object-oriented modeling techniques concentrate on the system's representation. By nature, those frameworks address a distinct, but related, aspect of system modeling. However, for the sake of completeness, we introduce three main languages that, at some points, have tended to interconnect with the software architecture modeling.

An early approach was defined by van Lamsweerde *et al.* with its *Knowledge Acquisition in Automated Specification of Software* (KAOS) [van Lamsweersde et al., 1991; Dardenne et al., 1993; Darimont et al., 1997]. KAOS empowers formal specifications of, among others, *goals*, *constraints*, *agents*, or *objects*, with formal links between those concepts. KAOS enables to verify requirement models regarding user-defined constraints, expressed as formal specifications. The KAOS framework is still evolving nowadays, integrating new concerns like *risk management* or exception handling [Cailliau and van Lamsweerde, 2012, 2014].

Another approach emerged somewhat later, called the i* framework [Yu and Mylopoulos, 1994; Yu, 1997]. This modeling and reasoning technique relies on two

[34] PRR is a vendor-neutral representation language for *business rules management systems*, i.e. software systems based on *business rules*, like for example legal regulation or credit contingency calculation, that can be abstracted and maintained separately from the software application code.

types of model: a *Strategic Dependency* (SD) and a *Strategic Rationale* (SR). The first type of model addresses relations between stakeholders, namely *actors*, in a particular organizational context. The latter focuses on the relations between those actors and a (software) environment. Like KAOS, formal definitions of relations between modeling elements may be stated and verified. Some research has tempted to link i* models to software architectures, but the semantic expressiveness of such system models was standing at a coarse-grained levels, *i.e.* mainly in terms of components and connectors [Gross and Yu, 2001]. The i* models have been integrated into a more general requirement framework named *Goal-oriented Requirements Language* (GRL), that also integrates aspects of late requirement engineering [Liu and Yu, 2004; Amyot et al., 2010; Marosin et al., 2014].

1.3.11 Lessons Learned from Architectural Knowledge Methods

In the present section, we crossed over significant research regarding design decisions with a special attention to architectural knowledge management. We did not aim at being exhaustive, other tools and techniques exists with similar concerns [Winkler and Pilgrim, 2010; Tang et al., 2010]. The ARCHIUM tool, for example, focuses on keeping explicit the link between a *component-and-connector* model and the implementation code in Java [Jansen et al., 2007; Jansen, 2008]. Zimmermann *et al.* proposed a formal decision model framework with predefined relations between decisions and alternatives such as refinement, decomposition and incompatibility associations [Zimmermann et al., 2007, 2009]. In a workshop report, Avgeriou *et al.* identified that, among others design patterns are also valuable parts of the architectural knowledge [Avgeriou et al., 2007]. They offer reusable solutions to recurring problems if documented appropriately with their addressed issues and decisions [Harrison et al., 2007].

As observed in the survey as well as the pilot and case studies we mentioned in the current section, putting some effort into the documentation of the decision-making process is not useless. These studies highlighted some interesting benefits in knowledge communication, maintenance activities, and in a systematic exploration and recording of alternative solutions.

However, an important tradeoff must be found between the needed additional work and the expected return on investment. The right level of details should remain project-specific or even could depend on the type of the decision itself, i.e, how deeply the solution will affect the architecture or how critical the decision is for the stakeholders. We thus argue for a flexible knowledge management technique combining codification and personalization strategies where architectural artifacts are related to the addressed requirements with a semi-formal representation of the decision-making process.

1.4 Transformation Languages

The Model-Driven Engineering discipline highly count on model transformations to, among others, afford the excessive work of maintaining different representa-

tions of the same system. In the past years, many dedicated languages and tools have been developed to write and execute model transformations. Czarnecki and Helsen proposed a feature-based classification framework from an extensive survey they conducted on languages coming from the scientific literature, the OMG *Query-View-Transformation* (QVT) standard, implemented in open-source tools and from commercial products [Czarnecki and Helsen, 2003, 2006]. They analyzed the languages features concerning, among others, *directionality, application conditions*, definition and uses of *patterns* and *variable typing*. Mens *et al.* proposed a taxonomy intended to help developers to choose among the transformation approaches according to their needs [Mens et al., 2005]. The authors introduced two orthogonal dimensions to model transformation approaches, *endogenous/exogenous* versus *vertical/horizontal* and highlighted some properties they consider as important assets for such languages and tools.

As stated in Section 1.1.5, we concentrate on *model-to-model* (M2M) facilities and languages. In this field, we can identify four main paradigms: *imperative, declarative, graph-based* and more recently *concrete-syntax-based*. Hybrid approaches also exist that combine imperative and declarative constructs [Gardner et al., 2003; Tamura and Cleve, 2010; Kappel et al., 2012]. In the following section, we will illustrate all four paradigms with the OMG standard on *Query/View/Transformation*, the ATLAS Transformation Language, the *Graph Grammar* approaches and the *Model Transformation By Example* techniques. We will afterwards discuss the *reusability* problem of model transformations and what kind of approach we believe is the more appropriate to our current needs.

1.4.1 OMG Query/View/Transformation

At the early ages of the MDE and based upon their MDA standard, the *Object Management Group* wrote a *request for proposal* for *MOF 2.0 Query / Views / Transformations* [OMG, 2002]. From the eight initial proposals, the first adopted version of the *Query / View / Transformation* (QVT) standard was proposed in 2008 [OMG, 2008] and the actual version was adopted in 2011 [OMG, 2011b].

QVT is supported by three transformation languages: *Relations, Operational Mappings* and *Core*. The *Relations* declarative language defines rules for pattern matching and creation of templates. A source model is transformed into a target model, i.e. a relation is created between both models, with mapping rules and possibly depending on OCL expressions specifying **only** `when` this relation must hold and **every**`where` another condition also holds as a consequence. Complex mappings can be defined with structural *templates*. Listing 1.11 illustrates two simple `relations` to transform a UML class diagram into a *Relational DataBase Management System* (RDBMS) formalism where UML packages are mapped to schemas and classes to database tables.

```
1 relation PackageToSchema { /* map each package to a RDBMS schema */
2   domain uml p:Package { /* a 'Package' in the domain 'uml' */
3     name = pn /* has a name */
4   }
5   domain rdbms s:Schema { /* a 'Schema' in the 'rdbms' domain */
```

```
    name = pn /* receives the same name as the Package */
  }
}
relation ClassToTable { /* from UML classes to RDBMS table */
  domain uml c:Class { /* 'Class' definition with a specific template */
    namespace = p:Package {},
    kind = 'Persistent', /* the property 'kind' must have the value 'Persistent */
    name = cn
  }
  domain rdbms t:Table { /* definition of 'Table' the in 'rdbms' domain */
    schema = s:Schema {}, /* a table into the schema as defined above */
    name = cn, /* the table has the same name as the uml class */
    column = cl:Column { /* a column is created */
      name = cn + '_tid', /* '_tid' is concatenated to the name of the table */
      type = 'NUMBER' }, /* the column type is set to 'NUMBER' */
    primaryKey = k:PrimaryKey {
      name = cn + '_pk', /* a primary key is defined on this table */
      column = cl } /* that points to the created column */
  }
  when { PackageToSchema(p, s); } /* condition to invoke PackageToSchema */
  where { AttributeToColumn(c, t); } /* external relation that must hold */
}
```

Listing 1.11: UML Class to RDBMS Table in QVT-Relations (from [OMG, 2011b])

Operational Mappings is a hybrid declarative and imperative language where transformations are mainly expressed in an imperative language that can be extended with relational transformations. A simple example of a *Book* to *Publication* model is presented in Listing 1.12 where books with chapters are transformed into publications with their aggregated number of pages.

```
metamodel BOOK { /* definition of meta-model BOOK */
  class Book { /* a book is composed by chapters */
    title: String;
    composes chapters: Chapter [*];
  }
  class Chapter { /* a chapter contains an amount of pages */
    title : String;
    nbPages : Integer;
  }
}
metamodel PUB { /* definition of meta-model PUB */
  class Publication { /* pub. has an amount of pages */
    title : String;
    nbPages : Integer;
  }
}

/* declaration of operational transformation */
transformation Book2Publication(in bookModel:BOOK,out pubModel:PUB);
main() { /* main execution procedure */
  /* invoke mapping rule */
  bookModel->objectsOfType(Book)->map book_to_publication();
}
mapping Class::book_to_publication () : Publication {
  title := self.title; /* copy title name */
  nbPages := self.chapters->nbPages->sum(); /* aggregate number of pages */
}
```

Listing 1.12: Book to Publication example in QVT-Operational (from [OMG, 2011b])

The *Core* language is also declarative, like the *Relations* language, but matchings are directly defined on model elements in meta-models. It is then more verbose

since it requires to define explicitly the patterns, the creation and deletion of objects and the transformation traces (in a MOF-compliant model), all being implicitly created by in the *Relations* language.

Operational and *Relations* languages have been implemented, mainly on top of the Eclipse tool, like in the Eclipse M2M platform[35].

From the QVT standard, a high interest for transformation facilities was born and many other declarative or hybrid languages were proposed as alternatives to QVT, but often still compatible with the underlying MOF architecture.

1.4.2 ATLAS Transformation Language

The *ATLAS Transformation Language* (ATL) is a hybrid declarative and imperative language [Jouault and Kurtev, 2005; Jouault et al., 2008]. Models are expressed in OMG *XML Metadata Interchange* (XMI) [OMG, 2013b] format or in a dedicated text-based DSL named *Kernel Metameta-model* (KM3) [Jouault and Bézivin, 2006]. A sample transformation rule for a UML class to RDBMS table handling monovalued attributes only is illustrated in Listing 1.13.

```
1  rule Class2Table { -- handle class
2  from
3    c : Class!Class -- input model is a class
4  to
5    out : Relational !Table ( -- output is a table
6      name <- c.name, -- its name is the class's name
7      -- all monovalued attributes are added in the ordered list 'col'
8      col <- Sequence {key}->union(c.attr->select(e | not e.multiValued)),
9      key <- Set {key} -- unordered set of retrieved keys
10   ),
11   key : Relational!Column ( -- definition of the key
12     name <- 'objectId',
13     type <- thisModule.objectIdType
14   )
15 }
16 rule ClassAttribute2Column { -- handle monovalued attributes
17 from
18   a : Class!Attribute (
19     -- any OCL function can be used in ATL
20     a.type.oclIsKindOf(Class!Class) and not a.multiValued
21   )
22 to
23   foreignKey : Relational!Column (
24     name <- a.name + 'Id',
25     type <- thisModule.objectIdType
26   )
27 }
```

Listing 1.13: UML class to RDBMS table in ATL (adapted from [Jouault et al., 2008])

A number of imperative extension can be used within ATL transformations. `Helper` functions can be defined in the context of a specific model object to compute an *attribute* or to create reusable pieces of code. Opposed to the above listing that presented declarative *matched* rules, a *called* rule is intended to be explicitly called to generate target model element from imperative code *block*. Imperative block can be called from the *action* block of a matched rule in combination with the target

[35] http://iot.eclipse.org

pattern (after the keyword `to`) or in the body of a called rule. For all transformed element, *traceability links* are automatically created in the transformation engine and bind the source and target elements with the involved rule.

The execution order of mapped rules is nondeterministic since the rules are called as the transformation engine navigates through the source model. ATL transformations are unidirectional, from a single *read-only* source model to possibly multiple *write-only* target models. Thus, no change may be applied on the source model neither the target models may be crossed during a transformation. Model merging is then not possible in ATL too. *Persistent* transformations, i.e incremental model transformation with change propagation, is not fully supported in ATL so any manual change made in the source or target models are not preserved after a transformation [Tratt, 2005; Jouault and Tisi, 2010].

The language is supported by an Eclipse plugin composed by a dedicated language editor, a compiler and a virtual machine, and is part of the aforementioned M2M platform [Jouault et al., 2008]. According to the authors, the ATL transformation engine is widely used in the academic and industrial worlds.

1.4.3 Graph Grammars

Aside the programmatic declarative and imperative approaches, graph grammars are becoming a popular field of study for model transformations. Graph grammars were first introduced to handle picture processing problems with *web grammars* by Pfaltz and Rosenfeld [Pfaltz and Rosenfeld, 1969]. Independently, Schneider proposed *n-diagrams* to give a formal definition for programming with diagrams and transformation of programming languages [Schneider, 1971]. The main purpose of graph grammars is to parse and generate graphs through production rules. Graph grammars are based on strong theoretical basis, mainly algebraic or set theoretic [Nagl, 1987; Fahmy and Blostein, 1992]. They may apply to various aspects of the software engineering practice from code optimization, formal semantics analysis or database engineering [Nagl, 1978]. Many graph grammar approaches have been developed in the model transformation field, like *Algebraic Graph Grammar* (AGG) [Ehrig et al., 1973], or *Triple Graph Grammar* (TGG) [Schürr, 1995].

The starting point of TGG is a proposal made by Pratt with its *pair grammars* that represent links between *languages*, especially from a concrete program syntax to its abstract syntax defined as a directed graph [Pratt, 1971]. Based on Pratt's proposition, Schürr introduced the notion of *Correspondence Graph* to face the lacks of *graph rewriting* or *graph grammars* approaches that were mainly restricted to *instance-specific* transformations [Schürr, 1995]. TGG is rule-based, purely declarative and relies on a formally-defined mathematical basis. Many implementations are still extensively developed with various expressiveness, scope, limitation and performance [Ehrig et al., 2005; Hildebrandt et al., 2013]. Figure 1.21 depicts a sample example that illustrates the general idea behind TGG approaches on the aforementioned *UML Class to RDBMS* example.

Bidirectional mapping must be defined with elements from both source and target models, with the creation or deletion rules between them. In Figure 1.21, `CT` and

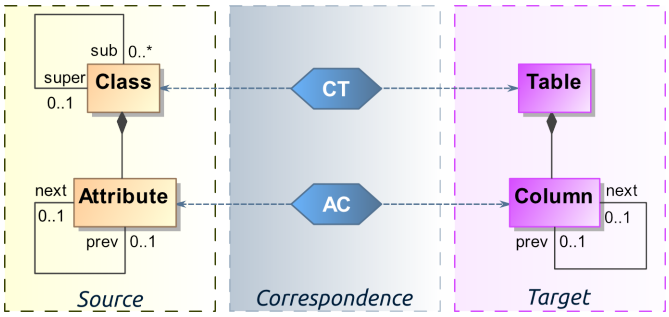


Figure 1.21: Overview of a *Triple Graph Grammar* for UML to RDBMS transformation (adapted from [Schürr and Klar, 2008])

AC denotes the *correspondence structures* from UML classes to RDBMS tables. Such a correspondence is composed by five *productions* (or *rules*) informally described as:

(1) creation of the root class with a new table
(2) association of a new subclass of an existing class with an existing table
(3) creation of the first attribute of a class together with the first column of a table
(4) creation of the first attribute and a *new* last column
(5) creation of a new last attribute with a new last column

In order to avoid invalid input processing and output creation, *Negative Application Conditions* (NAC) should be specified, i.e. a *pattern* found on the input graph forbids the application of the rule on the target graph. Figure 1.22 depicts the graphical representation of the fourth rule previously defined. The first NAC on the left states that the class may not have any attribute already defined. The second condition enforces to add the new column at the end, i.e. the created column has a *previous* one that cannot be followed by another column.

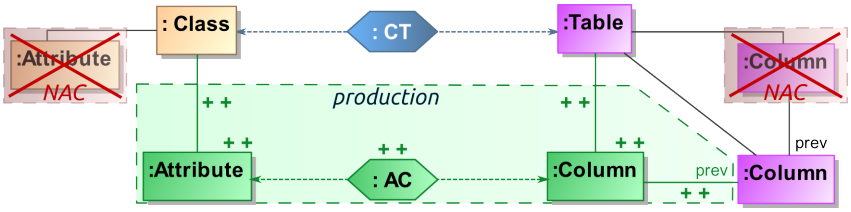


Figure 1.22: Sample rule with related NACs (adapted from [Schürr and Klar, 2008])

TGG is particularly promising for incremental and bidirectional transformations for models that can be represented as graphs, because *forward* and *backward* rules can be derived and executed separately if, among others, no concurrent modification is executed on the source and target model elements [Giese and Wagner, 2006; Schürr and Klar, 2008; Hildebrandt et al., 2013; Leblebici et al., 2014]. Furthermore, the native abilities for bidirectional links and incremental transformations are valuable assets to maintain traceability links between models produced by dif-

ferent tools [Königs and Schürr, 2006] or at different levels of abstractions [Rieke and Sudmann, 2012]. However, the average complexity of TGG approaches still remains high for some type of model transformations where operational semantics must be added to transformation rules and the lack of support for negative application conditions in some tools tends to accept or produce invalid models [Schürr and Klar, 2008; Buchmann et al., 2009]. Furthermore, with purely declarative approaches, backtracking may be needed to generate a target model, but very recent proposals tend to tackle this problem and now enable advanced concurrent model synchronization [Hermann et al., 2012; Gottmann et al., 2013].

1.4.4 Model Transformation By Example

Recently, a new trend in the model transformation field consists in deriving semi-automatically transformation rules from source and target example models [Varró, 2006; Wimmer et al., 2007]. *Model Transformation By Example* (MTBE) is inspired by similar approaches in the database domain with *Query-by-Example* generation [Zloof, 1975] or for (semi-) automatic generation of *eXtensible Stylesheet Language Transformations* [W3C, 2007] (XSLT) [Ono et al., 2002]. MTBE is not really a different paradigm by itself since the generated code is either declarative or hybrid, but at the opposite of the aforementioned languages, transformations are defined on the concrete syntax directly.

Two main methods can be distinguished: *correspondence-* and *demonstration-based*. Correspondence-based approaches are somewhat similar to TGG, they use a *triple* containing an input model, its semantically equivalent output model and correspondence mapping rules. Current MTBE tools generate either graph transformations or ATL code [Kappel et al., 2012]. *Model Transformation By Demonstration* (MTBD) approaches require the user to illustrate and annotate each transformation with input/output models so that transformation rules can be (semi-) automatically inferred from these examples [Langer et al., 2010]. MTBD usually need multiple iterations to fine-tune the examples, the generated transformation rules or both.

By definition, MTBE *by-correspondence* are limited to exogenous and horizontal transformations because both models must be semantically equivalent with the same level of details and correspondence rules are always expressed from one language to another [Kappel et al., 2012]. Their usage is then limited to model traceability and change propagation between models when they rely on bidirectional formalism, such as graph grammars. However, they are offering a user-friendly alternative to abstract syntax-based approach since there is no need for practitioners to learn a new language, either being familiar with the overall permitted construct and constraints of the language they manipulate. MTBD approaches have interesting assets for *test-driven development* of model transformations since they rely on example models. But this *by-example* technique is also one if their main drawback since users are often required to fine tune the transformation rules themselves, requiring them to dive into the abstract syntax of the modeling language. Finally, the lack of empirical studies in the MTBE field avoid to actually evaluate their benefits and weaknesses on real-world case studies.

1.4.5 Lessons Learned from Model Transformations

In the present section, we introduced the main paradigms of *model to model* transformation languages with some selected methods and tools. Many other languages and frameworks exist in the literature, such as MOLA [Kalnins et al., 2005], Kermeta [Muller et al., 2005a,b], GReAT [Balasubramanian et al., 2006], MT [Tratt, 2006], VIATRA [Ráth et al., 2008] or JTL [Cicchetti et al., 2010]. The present discussion and enumeration does not pretend to be exhaustive but to give an overview of the different approaches with their benefits and limitations and to highlight the common challenges to the model transformation practice.

Probably the first lesson to learn from the present study is that there is no such a best approach. The decision to use a transformation technique instead of another primarily depends on the domain and the intended usage. Declarative transformation rules, including graph grammars, offer a simple mean for pattern definition and cover the overall spectrum of the transformation taxonomy introduced by Mens *et al.* [Mens et al., 2005]. Besides their bidirectional and incremental abilities, *Triple Graph Grammars* are also very suitable to define traceability links between different types of models without affecting the source and target models [Königs and Schürr, 2006; Rieke and Sudmann, 2012]. However, they do not support rule inheritance, complex computation requiring an operational semantics and can consume much memory [Kusel et al., 2013a].

Reusability of model transformations is also becoming a popular research topic. Many languages integrate mechanisms to empower the definition of reusable *modules*, *rule inheritance* or *higher-order-transformations* [Tisi et al., 2009], i.e. transformations that use a transformation as input and produce another transformation as output [Kusel et al., 2013b]. However, concrete reusability should probably require some kind of standardization to tackle the heterogeneity of transformation and meta-modeling frameworks [Wimmer et al., 2010]. Recent research are also targeting to apply a *model driven development* method for transformations themselves in order to support the overall life-cycle of transformations [Guerra et al., 2010, 2013].

In our software architecture specific domain, as stated in Section 1.2.13, we are mainly interested in **(i)** traceability links between architecture models and addressed requirements, and **(ii)** endogenous horizontal and vertical model transformations. As argued in Section 1.3, patterns play an important role in software architecture for systematic reuse of best practices and architectural documentation purposes. We then need a mechanism close to the TGG and MTBE *correspondence-rules* to externally define structural patterns as reusable artifacts. Upon the heterogeneity in meta-models of current transformation engines, we do not believe that an absolute reusability *in the large* through complex higher-order-transformations or binding models is achievable in a *user-friendly* way. Abstract syntax-based transformations, especially for declarative rules, require software architects to learn an additional language and to manipulate more complex conceptual tools than structural concepts and *first-order predicate logic* properties. Thus, we do believe a concrete syntax approach is more suitable for our purpose and would raise the AK traceability without requiring much external documentation effort.

CHAPTER 2

ADRESSED PROBLEM AND RESEARCH QUESTIONS

We introduce now what are the main contributions of the present dissertation. We first recapitulate the problems we are addressing, as stated in the previous chapter. Afterwards, we formalize and discuss our research questions.

2.1 What kind of problems do we intend to solve?

In the previous chapter, we presented the main research in software architecture modeling techniques. We discussed about their modeling facilities as well as their expected benefits on academic research and on industrial sights. We especially compared their characteristics as communication means, model analysis facilities, requirement linking and design rationale traceability. As argued in Section 1.2.13, current software architecture modeling languages usually neglect the decision-making process that actually produced a specific architecture. Also, many of these languages do not provide flexible mechanisms to represent the communication between independent entities.

We crossed over notable work in the software architectural knowledge domain, from a formal ontology of design decisions to effective recording techniques evaluated in industrial case studies or actually used in the industry. We identified their advantages for software architecture design, maintenance and communication. As we discussed in Section 1.3.11, architectural knowledge with design rationale and alternatives must usually be recorded in external tools or must be retrieved by dedicated experts afterwards, these practices being highly time-consuming. Fur-

thermore, the expected level of details for documentation and traceability is usually project-specific or depends on companies' goals and practices. However, the need for appropriate recording of architectural knowledge is recognized by a significant amount of researchers as well as in polled industrial organizations.

We also illustrated the main trends in today's model transformation languages. We considered their intended advantages in the software architecture field, especially for model design and traceability. In Section 1.4.5, we talked about the reusability problem in current transformation approaches and we argued that a concrete-syntax-based transformation technique should suit more appropriately for endogenous model to model transformations where these transformations play a significant role in architectural knowledge recording, documentation and communication.

2.2 Research questions

We now formulate our contribution around three main research questions, refined into more concrete sub-questions.

2.2.1 RQ1 How can we model software architectures (SA) as effective means of communication ?

One of the main objective of an architecture model is to depict the structure of a software system in such a way it can be discussed between practitioners and communicated to other stakeholders. What kind of modeling constructs and techniques should we integrate in an architecture modeling language ? More precisely, we can refine this first research question into three sub-questions.

RQ1.1 How can we represent SA models at different levels of abstraction ?

Depending on the involved stakeholder, the expected level of details of a software architecture model should be configurable with some freedom, like offering a black-box only representation of the system or concentrating on specific user-defined requirements.

RQ1.2 How can we represent SA models with flexible communication facilities ?

Software architectures may be composed by lots of semantically different building blocks that may communicate through a wide range of technological alternatives. How such a flexibility can be included in an architectural language without drastically augment the number of modeling constructs neither the model complexity ?

RQ1.3 How can we represent SA models with deployment constraints ?

Target platforms and infrastructures play an important role in software development. Many physical constraints may have an impact at the architectural level, like the available network interfaces, the actual bandwidth, the storage disk space or the

processor power. How can all these constraints be modeled with regards to the software architecture in order to enable user-defined model checking capabilities ?

2.2.2 RQ2 How can we remember pieces of architectural knowledge for SA models ?

Besides structural models, the rationale are often an important piece of information to understand *why* the architecture has been designed as it is. Hidden behind a particular model, many alternative solutions have probably been explored and discussed. A significant part of the architectural knowledge is concentrated in the decisions concerning an architecture model. We particularly reformulate this questions into two sub-questions.

RQ2.1 How can we explicitly retain the link between SA models and related significant requirements ?

One of the prior objectives of a software architecture is to translate stakeholders' requirements into components or architectural elements that may be interconnected and collaborate to achieve the latter needs. However, as the system grows and evolves, the link between the architecturally significant requirements and the architecture model is often lost. What kind of mechanism can be used to enforce this link throughout the life of the system ?

RQ2.2 How can we document the decision-making and argumentation processes when designing SA models ?

As the system is specified, many alternative solutions and design decisions are taken. In order to have a more straightforward understanding of an architecture model, the argumentation of the reasons that produced that particular model are often valuable. How design rationale and alternatives can be captured in a reusable and non-intrusive manner ?

2.2.3 RQ3 How can we iteratively design SA models in a systematic manner ?

Model transformation techniques offer a systematic way to produce models. When used appropriately, they can automate some aspects of model development and allow formalized traceability between models. How can such techniques be integrated into a component-based architecture design method with the aforementioned goals in mind ? More specifically, we can expand this question into two sub-questions.

RQ3.1 How can we use model transformation techniques to iteratively refine SA models with new concerns ?

A common practice to tackle complex system requirements is to successively refine structural models from a high-level of abstraction to a finer-grained model. Aside,

reusable concerns can be *injected* into structural models in a systematic and reusable manner through model transformations. To what extend would it be possible to generalize this technique to any modification made to an architectural model to formally trace the model evolution ?

RQ3.2 How can we use model transformation techniques to build and document reusable architectural patterns ?

Consequently to the previously stated questions, patterns are key aspects of both architectural models and architectural knowledge. They define reusable pieces of blocks that address recurrent problems and from these patterns, architecturally significant requirements may be retrieved to enhance model understandability. How such definitions can be expressed in an individual, self-defined and reusable way for software architecture design ?

CHAPTER 3

A THREE-LAYER ADL, DEFINITION-ASSEMBLAGE-DEPLOYMENT

In this chapter, we present the structural formalism dedicated to represent system architectures in three specific layers: ***Definition****, a type-level definition of architectural constructs and styles,* ***Assemblage****, a concrete instantiation of a software architecture and* ***Deployment****, the mapping of the concrete architecture to infrastructure constructs. We also present the example case we use throughout this dissertation to illustrate our modeling formalisms.*

3.1 Language Overview

The work depicted in this dissertation is partially based on a custom *Architecture Description Language* (ADL) for component-based systems. We first introduce the main principles and objectives of our ADL and explain the reasons why we decided to adopt a layered representation. We then reproduce the language meta-model that we detail and discuss in the subsequent sections.

3.1.1 Main Principles and Objectives

In Section 1.2, we identified the main assets and shortcomings of actual component-based modeling approaches. Upon the consensus of using *components and connectors* syntactical constructs, a wide range of semantically different definitions were developed, especially regarding the interfaces and the types of connections.

The definition of architectural styles and patterns is important to enhance the reusability of solutions to common problems. Styles and patterns are also valuable as architectural templates in early design phases. An architectural language should thus enable to separate a type of architecture to its instances.

Furthermore, a component may be instantiated more than once, or it can be created or deleted by other ones. This piece of information usually needs external behavior specifications or relies on a specific deployment and runtime platform, so that it cannot be expressed on the structural model itself.

Deployment constraints are scarcely addressed by existing approaches, especially during the overall design phase. It is not uncommon to know infrastructure-related constraints very early in a software development process and an explicit recording of these constraints related to the logical components can help to identify incompatibilities as soon as they arise.

As pointed in Section 1.2.13, the semantics of *communication links* are mainly either restrictive (like synchronous operation calls or value passing, for example), must be defined by dedicated behavioral specifications in another formalism (like UML statecharts, for example) or need to be selected among many different modeling elements. More flexibility in connector definition should be found without leading to an unacceptable level of complexity or an explosion of the amount of modeling possibilities.

3.1.2 Why a Layered Representation ?

From these observations, we decided to propose a *three-layer* ADL that addresses these three concerns: *types* of architectural elements and styles, possible running *instances* and *deployment* mapping rules on hardware/software platforms. We use a sort of *layered* representation to separate these three aspects, but to make possible to describe all of them with their inter-relations.

We chose to provide a separate layer to specify abstract modeling element *types* and possible structural configurations using these *types*. These element *types* and architectural patterns or styles can then be exported and reused in a convenient manner since they are specified independently. Abstract computation and communication modeling elements can also be defined with user-defined properties. These constructs are used to depict a physical configuration of machines and network paths and will be particularly useful for the *deployment* mappings. These definitions can be externalized in dedicated libraries and reused across models.

From these *types* and patterns, one or more concrete configurations can be defined. One architectural style can be instantiated in several ways, depending on the number of concrete *instances* of each component, for example. An abstract

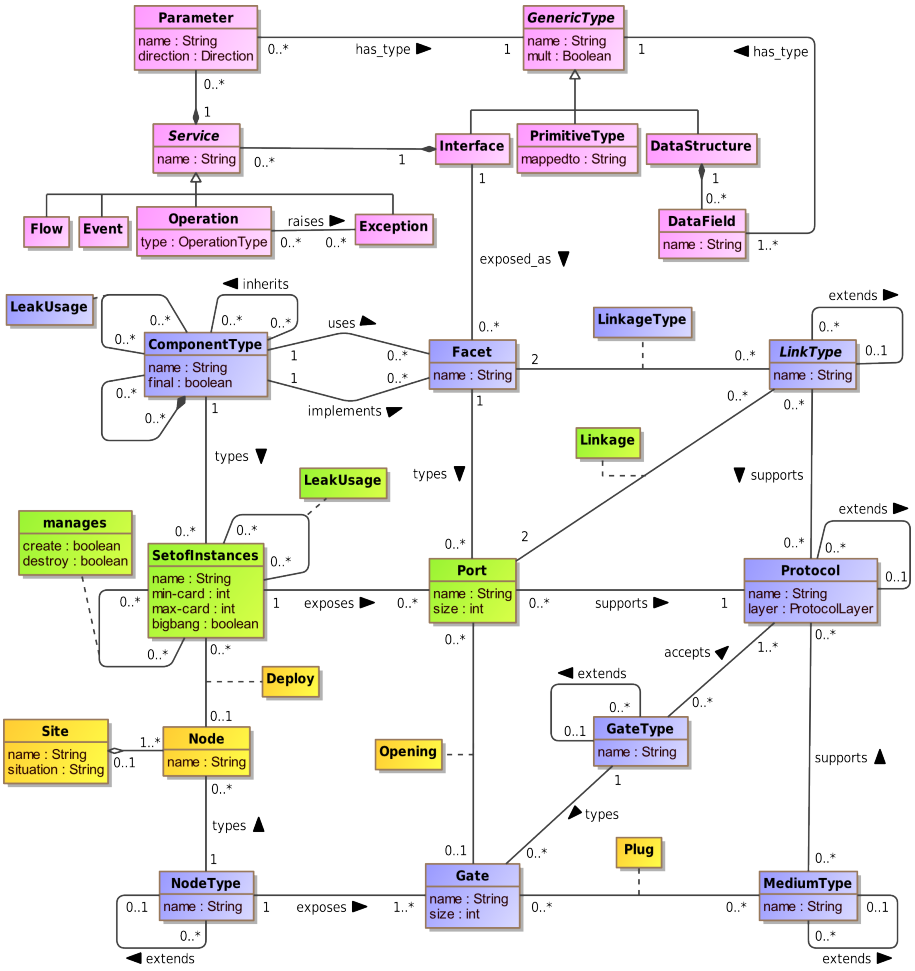


Figure 3.1: **Definition-Assemblage-Deployment** meta-model

connection between types of components may also be concretely materialized by many concrete communication protocols.

The *instances* can be mapped to abstract representations of a *deployment* infrastructure with user-defined properties. Such a mapping may help at identifying incompatibilities between a possible instantiation and the available computing and communication material. Alternatively, needed properties for a particular configuration can also be specified to define runtime constraints very soon during a development project.

In some ways, the separation between *types* and concrete *instances* is close to the *Platform Independent Model* and *Platform Specific Model* as introduced by the OMG *Model Driven Architecture* approach [ORMSC, 2001], the *instances* being specific to the platform depicted in the *deployment* mapping.

3.1.3 Language Meta-Model

We now introduce our formalism dedicated to depict structural models for software system architectures. It is called **Definition-Assemblage-Deployment** (**DAD**), each term identifying one of the aforementioned *layer*. All layers can be defined separately, but are dependent on the upper one, so that a **Deployment** depends on a particular **Assemblage** of modeling elements defined at the **Definition** layer. The meta-model is illustrated in Figure 3.1. The violet and blue elements belong to the **Definition** layer, the green ones to the **Assemblage** and the orange elements to the **Deployment**.

3.2 Definition

The Definition layer can be separated into three types of modeling elements: the *Types*, the *Structural Model Elements* and the *Running Infrastructure*.

3.2.1 Types

The *Types*, depicted in Figure 3.2, aim at defining the types of data that can be exchanged in a software system, as well as how they can be exchanged.

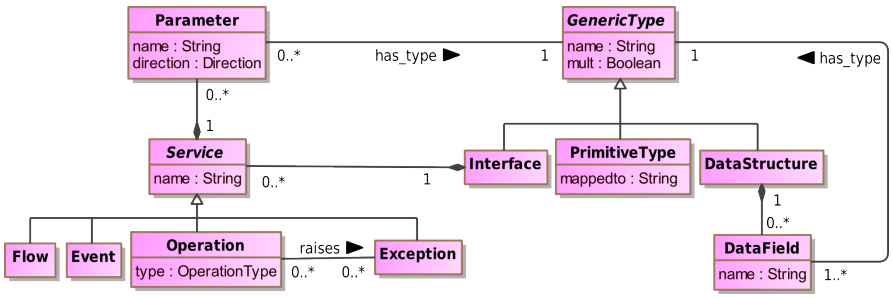


Figure 3.2: *Types*-related modeling elements

`GenericType`

A `GenericType` is an abstract object used to type `Parameters`. It may be a `PrimitiveType`, a `DataStructure` or an `Interface`. It has an identifying `name` and a boolean value named *mult* saying if the type is multivalued.

`PrimitiveType`

A `PrimitiveType` is a data type that may hold values of the same type only. It can be *mappedto* another type defined elsewhere (referred with its full name). At present time, Java classes can be referenced and imported as `PrimitiveTypes`.

DataStructure

A DataStructure is a complex type containing an ordered list of heterogeneous DataFieds. This list may be empty.

DataField

A DataField is a data variable contained in a DataStructure. It holds a value of a certain GenericType, that GenericType may be multivalued, expressed by the *mult* boolean flag.

Interface

An Interface is a special type that contains Services. It gathers semantically-correlated interaction points between components in a software system. The list of Services may be empty.

Service

A Service is an interaction point identified by a *name* that may be used to provide some visible effect, like exchanging data Flows, producing Events or executing Operations that may raise Exceptions. A Service contains an ordered list of Parameters of given GenericTypes, this list may be empty. All these types of Services empower to abstractly represent a wide range of communication standards or technologies like for example, CORBA [OMG, 2012b], REST architectures [Fielding, 2000] or other types of web services. They will be detailed in the following paragraphs.

Parameter

A Parameter is a variable of a certain GenericType that may hold a value. It is contained in a Service. It has also a *direction* attribute saying how it is handled by the Service. The Direction type is an enumerated type with the following values:

in	input parameter (read-only)
out	output parameter (write-only)
inout	combined input and output parameter (read-write)
return	output value

Flow

A Flow is a special type of Service that provides output values of a single *type*. A Flow may supply only one *out* Parameter, but the parameter may be multivalued. A **DAD** Flow is analogous to a SysML item flow and is mainly used to represent *value-passing* channels, like gas, water or emergency transmitters.

Event

An Event is a Service that is triggered by an internal stimulus and performs some behavior with a visible effect, i.e, some internal condition activates some behavior that produces a structured message. An Event may contain *in* Parameters only. As examples, interrupt signals in operating systems or signals from peripherals may be represented as Events.

Operation

An Operation is a Service that performs some behavior with notable or visible effects from external calls, i.e, is activated from the outside of its owner object and has a meaning in terms of its behavior for a software system. An Operation has a *type* saying if it is *synchronous* or *asynchronous*. A Parameter in a *synchronous* Operations may be of any direction, but *asynchronous* Operations may only have *in* Parameters. *synchronous* Operations may raise Exceptions. Typically, Operations are program calls.

Exception

An Exception is a special structured message triggered from the execution of a *synchronous* Operation. It indicates that the normal execution flow of the associated behavior encounters an unexpected condition. Exceptions may have *in* Parameters.

3.2.2 Structural Models

Structural models are used to define configurations of component types connected by link types in order to form an abstract architecture model. Figure 3.3 concentrates on these modeling elements.

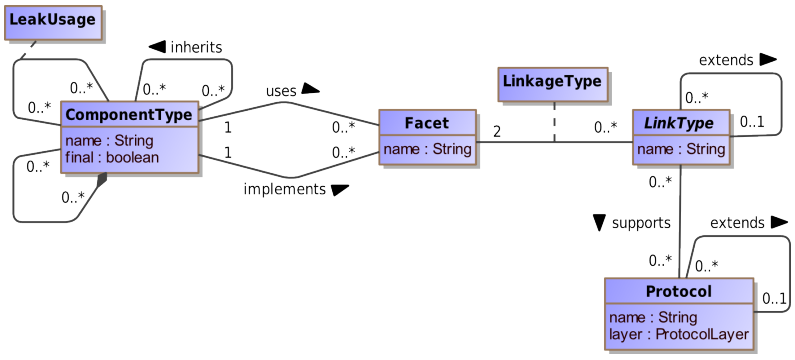


Figure 3.3: Structural modeling types

ComponentType

A ComponentType is an identifiable model element with computational capabilities that may use and/or implement Facets. ComponentTypes can be either connected to each other through their Facets with a LinkType or through a LeakUsage dependency when the precise semantics of this dependency is not known. This type of dependency has two main goals. First, it lets more freedom to modelers to draw an architectural draft at early design stages when the precise semantics of the bindings between components is not defined yet. Second, it is useful to define dependencies to external components, such as APIs or software libraries.

A ComponentType can be *composite*, so that it is composed by other *composing* ComponentTypes that can be likewise connected to each other using their Facets or LeakUsage dependencies and form an *inner-configuration*. Any ComponentType can be recursively refined into *inner-configurations*: a *parent* ComponentType may have *children*, that may be *parents* of other *children*, and so forth. The Facets of the *composite* ComponentType are named its *external* Facets and all Facets defined by its directly *composing parts* are named its *internal* ones. The same ComponentType may be child of multiple parents. It must be defined once as a *standalone* ComponentType and reused in multiple *inner-configurations*.

A ComponentType may *inherit* from one or more other ComponentTypes. The inheritance relationship means that the sub-ComponentType gathers all external Facets from its super-ComponentTypes and may add other *external* Facets. It may also overwrite its *inner-configuration* completely, i.e its internal representation may be defined differently than its super-ComponentTypes *inner-configurations*. Analogously to the Java programming language, it may be tagged as *final* to disallow further extensions of its definition. Since the inheritance mechanism delete and replace the *inner-configurations*, one may want to restrain the usage of a ComponentType in order to guarantee its behavior.

Facet

A Facet is the *materialization* of an Interface inside a ComponentType, i.e, the Interface is made available as an access point for the ComponentType. A Facet has a name that identifies it locally to its parent ComponentType. The Interface is either *used* or *implemented* by a ComponentType. A ComponentType that *uses* a Facet denotes a *require* dependency, *i.e.* the ComponentType needs the Interface referenced by the Facet. At the opposite, a ComponentType that *implements* a Facet denotes a *provide* offer, *i.e.* the ComponentType implements the behavior described in the Interface typed by the declared Facet. This type of dependency is called the *polarity* of the Facet. Facets are meant to connect ComponentTypes to each other either through *provide-require* or *delegation* contracts via Linktypes. *Provide-require* contracts connect Facets of opposite polarities. *Delegation* contracts connect two Facets of the same polarity and are used in *composite* ComponentTypes. It indicates that the behavior of an external Facet is effectively transmitted to an internal Facet.

Protocol

A Protocol is used to specify the properties that may be used to support the communication between structural modeling elements. A Protocol has an identifying name and its semantics may be refined by user-defined properties. The Protocol is used to specify the behavior of the communication and makes it independent from the way the ComponentTypes are linked to each other. It also indicates on which communication *layer* it operates. The communication layer is a mandatory feature for the Protocol because it will be used to ensure that the connection between ComponentTypes' instances is effectively possible. Furthermore, we want to provide enough flexibility to represent many types of connections between any type of components (software and hardware) using any type of communication technology. As we will detail through this Chapter, the Protocol has a key-role in the overall communication mechanism of **DAD** models

As a basic implementation, we extended the 7-layer *Open Systems Interconnection* (OSI) model [ISO, 1994] with one more possibility to express the communication between programs sharing the same memory space, like in the same Java virtual machine for example. We use the OSI as our reference model to be as generic as possible and to enable to represent a wide range of existing communication protocols. Alternative layered models, like the IEEE 802.X protocols, as the *Local Area Network* standard [IEEE, 2013] or TCP/IP protocols [Braden, 1989], can be mapped to the OSI layers. The Bluetooth® specification defines its own protocol stack [Bluetooth SIG, 2013]. Even if the correspondence to the OSI model is not as straightforward as for the other two layered models, such a mapping is still possible. All these representations are specific to some technologies or protocols and we would rather stick to an abstract model. This layering is expressed as properties such that any other model can be defined and used as Protocol properties.

A set of general purpose properties has been defined as a common *library*, with among others, *security*-related attributes. An extensible mechanism that is further detailed in Section 3.5, has been included into the **DAD** framework to specify user-defined properties.

LinkType

A LinkType is an abstract object that connects two Facets to each other. The same LinkType may be used in more than one connection, but for a given connection, a LinkType may not connect more than two Facets. The LinkType specifies the characteristics of the binding of two Facets, so ComponentTypes, but does not specify the orchestration of the communication itself, as the Protocol is intended to do. A taxonomy of LinkTypes is presented in Figure 3.4.

LinkTypes are either ConnectorTypes or DelegationTypes. A LinkType specifies the binding *pattern* between two Facets, *i.e.* how the instances of these Facets will be concretely connected in an **Assemblage** specification[1]. A ConnectorType will be used to connect two Facets of opposite *polarities*, from the Facet that *re-*

[1] This feature will make more sense in Section 3.3 when we talk about the **Assemblage** layer.

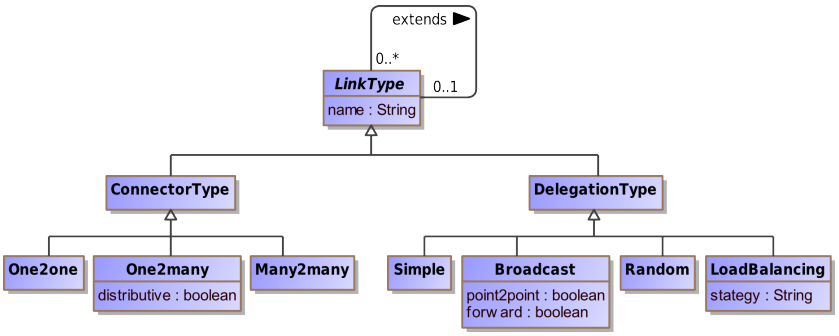


Figure 3.4: Built-in `LinkType` taxonomy

quires the `Interface` to the `Facet` that *provides* it, *i.e.* in a *provide-require* contract, using the following patterns:

`One2one` This is a *point-to-point* connection with one required instance connected to exactly one providing instance. All kind of `Services` with all types of `Parameters` are allowed on such `ConnectorTypes`.

`One2many` one required instance is connected to all providing instances of the bound `Facet`. If the `ConnectorType` is *distributive*, then the connection is a point-to-point connection between the required `Facet` to each of the providing instances. In this cas, *output* and *return* `Parameters` as well as `Exceptions` for `Operations` are accepted. In case of a *non-distributive* connection, only *input* `Parameters` are allowed. `Event` and `Flows` are also unrestrictedly accepted as type of `Services` for `One2many Connectors`.

`Many2many` many instances from the required side are connected to multiple ones on the providing side. It may be any number of instances, from none of them to all of them on each side. The same restrictions regarding the `Services` as the ones expressed for *non-distributive* `One2many ConnectorTypes` apply here.

A `DelegationType` is used to connect two `Facets` of the same polarity, from a *facade* `Facet` of any polarity to a *delegate* of the same polarity. A `DelegationType` is used to transfer the behavior defined in an `Interface` and exposed by a `Facet` from a *parent* `ComponentType` to another one from a *child* `ComponentType`. `DelegationTypes` are also refined into a set of subtypes to specify the linkage around the following patterns:

`Simple` simplest delegation between the *father* instance to one of its *children* `Facets`. Only one instance may be defined on both side of the `DelegationType`. Similarly to `One2one ConnectorType`, this type of connection is point-to-point and, like all other `DelegationTypes`, it accepts all type of `Services`.

`Broadcast` one instance delegates the `Facet`'s behavior to all its *delegate* instances either in a *one-to-many* or in a *point-to-point* connection

depending on the value of the *point2point* boolean attribute. The *forward* attribute specifies the number of Events, Flows or answers to synchronized Operations that will be processed from delegates. If forward is set to *true*, all delegates are intended to act the same way, *i.e.*, they have to all produce the Events, Flows or compute the return value before the Service is actually activated in the facade. If any delegate produce a different behavior to the same Service, it is discarded from the delegate. In the worst case, if all delegates produce a different behavior, the first to answer is kept as the unique delegate. If *forward* is set to *false*, the first activation or answer from a delegate is sent back to the facade.

Random one delegate is randomly chosen to execute the called Service. Each call may be redirected to a different *delegate*, or not. No assumption on the order of the selected instances either on a fair distribution of the workload between the delegates may be taken without adding more user-defined properties to the DelegationType. A Random DelegationType only ensure that a call will be transmitted to one and only one delegate.

LoadBalancing more elaborated connection between a Facet and its *delegates* may be defined with the LoadBalancing DelegationType. User-defined strategies can be specified at least by a *strategy* name. Any other DelegationType strategy is then considered as a LoadBalancing.

A LinkType may extend another LinkType. Even if we represented the *extends* relationship on the LinkType modeling elements for readability reasons, a concrete LinkType may only extend another LinkType of the same *concrete* type. For example, a One2one ConnectorType may only extend another One2one ConnectorType and the same rule applies to all other concrete LinkType.

When extending a LinkType, the subtype may not restrict the list of accepted Protocols, such that it does not accept a Protocol supported by its *supertype*. Besides, it may overwrite any other user-defined property.

LinkageType

At this level, types of architecture configurations, *i.e.* styles or patterns, can be expressed with LinkageTypes, depicted as the highlighted relationships between a LinkType and a Facet in Figure 3.3. A LinkageType always binds a *source* Facet to a *target* Facet with a LinkType. The polarities of the source and target Facets will depend on the concrete LinkType used, from a required to a provided Facet for ConnectorTypes and from the facade to its delegate for DelegationTypes.

LinkageTypes may not interconnect more than two Facets at a time, such that it is not possible to express a sort of *grouping* of connections between Facets that would model a complex interaction between components. This modeling choice has been made to somehow reduce the semantic uncertainty in such *n-ary* bindings that would require supplementary specifications in order to fully understand the

exact nature of the connection, *i.e* is there any priority between `Facets`, is there a *calling order* between two `Services` from interrelated `Facets` or does it express a specific policy or communication protocol, for examples.

3.2.3 Running Infrastructure

A set of construct types are dedicated to represent a particular *running infrastructure*. To this end, four modeling elements have been defined in the **DAD** language that are depicted in Figure 3.5 and detailed hereafter. All these elements are intended to be refined by user-defined properties to specify more precisely their semantics. The property definition mechanism is further detailed in Section 3.5.

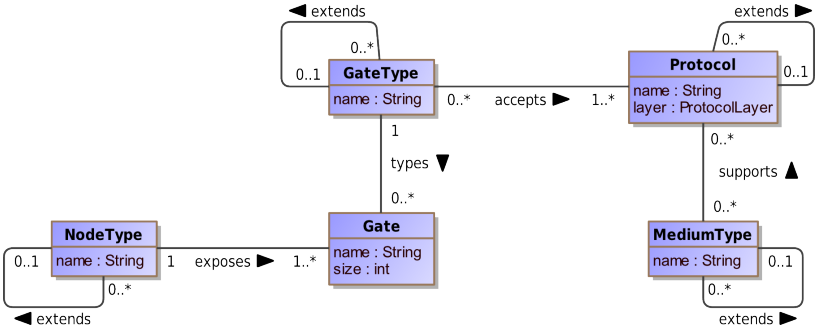


Figure 3.5: Physical infrastructure types

`GateType`

A `GateType` represents a type of physical communication port that can be plugged onto a hardware device or a platform environment. It is aimed to model the possible interaction points that can be placed into `NodeTypes`. It is identified by its *name* and may extend one other `GateType`. A `GateType` can accept a list of `Protocols`. It will be used to type `Gates` that are present in `NodeTypes`.

`NodeType`

A `NodeType` describes abstractly any type of physical hardware or software environment that may receive a piece of part of the architecture model to be run on it. It is identified by its *name* and may extend one other `NodeType`. It exposes a list of `Gates` that makes it reachable from other `NodeTypes`.

`Gate`

A `Gate` is the entry point of a `NodeType` for any type of communication. It is typed by a `GateType`. It is identified by a *name*. It also inherits the list of accepted `Protocols` and user-defined properties from its type. It can be replicated into a `NodeType`,

according to its *size* integer value, such that the Node contains multiple Gates of the same type.

MediumType

A MediumType is the concrete link that holds the communication between two Gates. It is identified by a *name* and supports a possibly empty list of communication Protocols. The same MediumType may be used to bind multiple Gates. It may also extend one other MediumType, but exactly as LinkTypes, it may not restrict the list of supported Protocols and may redefined any other user-defined properties.

3.2.4 Semantics of Model Element Extensions

We already discussed in detail the *extends* relationship for ComponentTypes, but not for the other modeling elements. Except for ComponentTypes which has a more complex extension mechanism, as presented in Section 3.2.2, all these other constructs may only extend one other construct. We detail here for each of them why we decided to allow only single inheritance relationships for these elements.

Extending more than one communication Protocol makes no sense to us since many incompatibilities between ProtocolAttributes can arise from such a multi-inheritance. As far as we know, we could not find any example of such a Protocol that combine orthogonal properties in one definition.

The LinkType may extend at most one other LinkType or the same type since the proposed taxonomy in Section 3.2.2 is orthogonal. Again, this makes no sense to extend, for example, a Simple and a Broadcast DelegationType at once because their communication semantics is not compatible at all. The same remark holds for all pairwise definitions of concrete LinkTypes.

Analogously to Protocols, GateTypes may only extend one other GateType. This choice is mainly motivated to avoid complex verifications and possible incompatibilities in the accepted list of Protocols when merging the definitions of multiple GateTypes.

Last, NodeTypes may extend only one other NodeType at a time. As presented in Section 3.2.3, NodeType semantics is intended to be refined by user-defined properties that may express, for example, the disk capacity, the processor power or even a running platform environment. We could not find any valid example of extending two or more hardware or software platforms (what would it mean if two CPUs have different computation speeds or architecture, for example), so we also decided to restrain the extension mechanism to a single model element. MediumType extension capabilities follows the same reasoning since the same kind of incompatibilities between properties may arise.

3.2.5 Semantics of ComponentTypes containments

As explained in Section 3.2.2, a ComponentType may recursively contain other ComponentTypes. In **DAD** models, the inclusion mechanism is *transitive*, i.e., if

a `ComponentType` has children that have children themselves, the grand-children are also considered to be included into their grandfather. Formally, if

- C denotes a `ComponentType`
- $\boxed{\subset}$ denotes the inclusion of a `ComponentType`, such that $C_c \subset C_p$ indicates that $\boxed{C_c}$ is a child of C_p

The inclusion is defined as a transitive relationship,

$$C_c \subset C_p \quad \wedge \quad C_g \subset C_c \quad \Rightarrow \quad C_g \subset C_p$$

Likewise, the reference to the direct parent `ComponentType` is denoted by the function $p(C)$, which is defined as follow:

$$p(C) = \begin{cases} root, & \nexists\, C_p \mid C \subset C_p \\ C_p \mid C \subset C_p \wedge \nexists\, C_c \mid (\, C_c \subset C_p \wedge C \subset C_c\,), & otherwise \end{cases}$$

Both these *inclusion* mechanism and *parent* function will be particularly useful in Chapter 5 when we will talk about **DAD** model transformations.

3.2.6 Qualified Names and Visibility

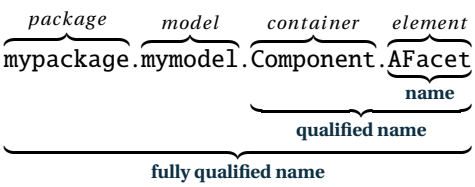
In **DAD** models, we use *qualified names* to identify elements through models. As we will illustrate in Section 3.7, a model belongs to a *package*. A *package* is simply a directory where all correlated models can be placed. *Packages* can be nested in other *packages*, exactly like directories on a file system or Java packages, so that a hierarchy of *packages* can be created.

From this structure, we can identify three types of names for any model elements:

name name as specified in the *name* attribute of the element
qualified for nested elements (like `Facets`) contained in other elements
fully qualified complete name with all packages and model names as prefixes

All names that compose a (fully) qualified name are separated by a *dot "."*, like in the Java programming language. This simple mechanism allows to always unambiguously identify any model element.

For example, for a `Facet` `AFacet` in a `Component` `ComponentType` (its *container*) that is contained in the `mymodel` placed in the `mypackage` directory, its names will be of the form:



Along with this naming convention, the visibility and uniqueness of a model element depends on its fully qualified name. All elements defined inside the *scope* of

another element, typically children `ComponentTypes` and `Facets`, must always be referenced by their qualified names. This mechanism offers some naming flexibility for inner elements with the same semantics. For example, the same `Interface` implemented in multiple `ComponentTypes` may be exposed with the same name for convenience. Since the `Facets` are contained in other `ComponentTypes`, their qualified names will then be different. An element is never visible outside his model, i.e. *scope*, except as otherwise imported[2].

3.2.7 Validity of `LinkageTypes` between `Facets`

In order to evaluate the compatibility of two `Facets` linked by either a `ConnectorType` or a `DelegationType`, two different types of checks are performed. First, both names are checked. If their `Interface`'s qualified names are identical, then the `LinkageType` binds two identical `Facets`, so the binding is obviously valid. In case of different names, a signature-based check is performed for all `Services`. The following compatibility verification rules offer more flexibility to combine components that can be defined by different organizations or for different contexts. We specify a kind of *duck-typing*[3] for **DAD** `Interfaces` to allow loose coupling between `Interfaces` as well as the possibility to combine different *middlewares* or communication technologies, especially like web services [Leff and Rayfield, 2010]. This way, at the moment a service provider offers at least the required `Services`, such a binding is possible.

Informally, we verify for the `Services` of the source `Facet` if we can find a `Service` of the same type with the same `Parameters` (without considering the names) in the same order in the target `Facet`. Each time a compatible match is found, the `Services` are removed from the inspected lists. A match must be found for all required `Services`, but the set of provided `Services` can be larger. The `Exceptions` that are raised from a `Service` are also verified, but the check is more stringent in this case. All `Exceptions` raised by the provided `Service` must be taken into account by the required `Facet`. More formally, let:

- p denotes a `Parameter`, such as $p = (d, g)$ with d the parameter direction and g its `GenericType`
- e denotes an `Exception`
- s denotes a `Service`, such as $s =< p_1, ..., p_n >< e_1, ..., e_m >$, an ordered list of `Parameters` followed by an ordered list of `Exceptions`.
- I denotes an `Interface`, such as $I = (s_1, ..., s_m)$, a list of `Services`
- F denotes a `Facet`
- $\models$ denotes the *is typed by* relationship, such as $F \models I$
- $F \in C$ denotes that the `Facet` F is exposed by the `ComponentType` C
- $\xrightarrow{L}$ denotes the directed `LinkageType` between `Facets` with a `LinkType` L

[2] Model imports will be detailed in Section 3.6.2

[3] Duck-typing is a dynamic typing technique in object oriented programming where the semantics of a method relies on its signature, instead of being specified by the inheritance of a particular interface or class. It has been named that way in reference to the poet James Whitcomb Riley for his well-known quote *«When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.»*

- $\boxed{\frown}$ represents a `LeakUsage`, such that $\boxed{C_1 \frown C_2}$ denotes C_1 uses C_2

For two `Parameters` $p1$ and $p2$, they are considered equivalent, denoted by $\boxed{\approx}$, if formally,

$$\begin{aligned} p_1 = (d_1, g_1) \quad &\wedge \quad p_2 = (d_2, g_2) \\ p_1 \approx p2 \quad &\Leftrightarrow \quad d_1 = d_2 \wedge g_1 = g_2 \end{aligned}$$

For two `Service`, s_1 and s_2, they are considered as equivalent, denoted by $\boxed{\approx}$, if formally,

$$\begin{aligned} s_1 =< p_1^1, ..., p_1^n >< e_1^1, ..., e_1^m > \quad &\wedge \quad s_2 =< p_2^1, ..., p_2^n >< e_2^1, ..., e_2^p > \\ s_1 \approx s_2 \quad &\Leftrightarrow \quad \forall p_1^k \in s_1, \exists\, p_2^k \in s_2 \mid p_1^k \approx p_2^k \\ &\wedge \quad \forall e_1^i \in s_1, \exists e_2^j \in s_2 \mid e_1^i = e_2^j \end{aligned}$$

Then, let C_1, C_2 two `ComponentTypes` and I_1, I_2 two `Interfaces`. If a `LinkageType` has been defined between two `Facets`, the following conditions must hold:

$$\begin{aligned} \forall\, F_1 \in C_1, F_2 \in C_2 \quad &\mid \quad F_1 \models I_1 \wedge F_2 \models I_2 \\ F_1 \xrightarrow{L} F_2 \quad &\Rightarrow \quad \forall\, s_1^i \in I_1, \exists\, s_2^j \in I_2 \mid \quad s_1^i \approx s_2^j \end{aligned}$$

We do not consider type compatibility in the above definition since **DAD** types may not be extended, *i.e.* we do not provide mechanism to define a hierarchy of datatypes. Again, this extension could be envisioned in a future release of the language and the $\boxed{\approx}$ relationship can be modified accordingly.

3.3 Assemblage

The second layer of **DAD** models concentrates on the definition of valid instantiations of architecture constructs and/or configurations specified at the **Definition** layer. As discussed in Section 3.1.2, we separate the type of architecture to its possible instantiations for two main purposes. First, some *dynamicity*-related attributes, such as the amount of running instances, are specified at this level, independently from the abstract definition of an architecture. Second, the concrete binding between instances must rely on one specific communication *Protocol*, where `LinkageTypes` at the **Definition** layer concentrate on the architectural topology.

The **Assemblage** modeling elements and rules are depicted in Figure 3.6. Some elements from the **Definition** layer are also shown on the figure to highlight the relations and *reuse* between both layers.

We present in the remaining of this Section all **Assemblage** modeling elements and discuss the verification rules of the concrete configurations specified at this layer.

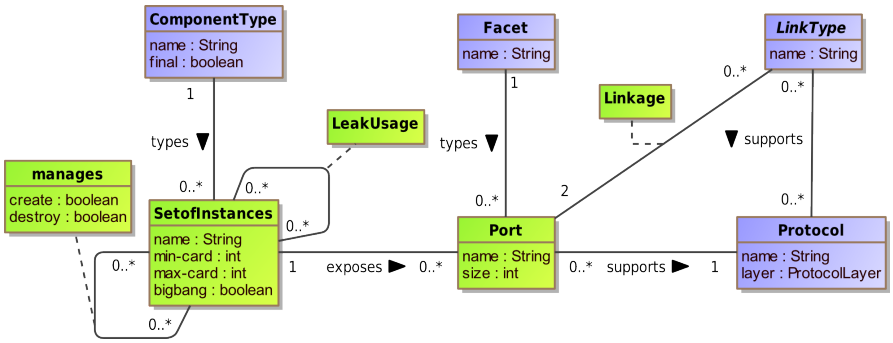


Figure 3.6: **Assemblage** modeling elements

SetOfInstance

A SetOfInstances is the *materialization* of a ComponentType in a concrete architecture. It is typed by a ComponentType, identified by a *name* and has a minimum (*min-card*) and a maximum (*max-card*) cardinality. These cardinalities express the amount of instances of the same ComponentType that may run simultaneously. Combined to user-defined properties, they can be useful to clearly state the conditions under which these properties hold. Furthermore, they will be particularly profitable for the **Deployment** layer where the mapping between software components and a target platform infrastructure can be defined.

A SetOfInstances may be flagged as *bigbang* to indicate that it must be manually instantiated. The *bigbang* flag can be seen like the `main` method holder class for a Java application. A SetOfInstances may *manage* other SetOfInstances to *create* or *destroy* them and may be managed by multiple ones.

It may use other SetOfInstances in an analogous way as the LeakUsage dependency for ComponentTypes. At this level, this dependency is particularly useful when using *Components Off The Shelf* (COTS) or other types of reusable libraries. It exposes zero or more Ports that define its *entry-exit* communication points. Finally, it can be refined by user-defined properties to add software-related constraints on, for example, memory or CPU usage, minimum version number of underlying libraries and so forth.

Port

A Port is a communication point at the boundaries of a SetOfInstances. It is typed by a Facet and identified by a *name*. It can be compared to an interaction point between pieces of software. A Port may be replicated according to its *size* attribute, so that it offers multiple entry points to the same Interface's behavior. A Port supports one and only one Protocol because a particular **Assemblage** expresses one possible instantiation of a configuration specified at the **Definition** layer. Ports may be interconnected via LinkTypes.

Linkage

A Linkage is the concrete connection between two Ports with one LinkType. As LinkageTypes, a Linkage is directed from the Port typed by a requiring or facade Facet to the providing or delegate one. We reuse directly the LinkType modeling element and do not instantiate individually each LinkType to create a Linkage. Requiring modelers to create a link for each of these bindings would have been counter-productive and verbose. Furthermore, we are only interested in the properties associated to the connection facility used to bind two SetOfInstances via their respective compatible Ports.

The validation of Linkages is analogous to the one for LinkageTypes, instead that for a given Port we have to retrieve its typing Facet and we have to ensure that the chosen Protocol for the Port belongs to the accepted list of Protocols for the LinkType actually used. Formally, let

- S denotes a SetOfInstance, being a set of Ports
- P denotes a Port and $\boxed{P \in S}$ denotes that P is exposed by S
- $\boxed{\models}$ denotes the *is typed by* relationship, such as $\boxed{S \models C}$
- T denotes a Protocol
- L denotes a LinkType
- $\boxed{\vdash}$ denotes the Protocol support, such that $\boxed{P \vdash T}$ and $\boxed{L \vdash T}$

If a Linkage exists between two Ports with a LinkType L, noted by $\boxed{\overset{L}{\frown}}$, the following conditions must hold:

$$
\begin{aligned}
\forall\, P_1 \in S_1, P_2 \in S_2 \quad | \quad & P_1 \models F_1 \wedge P_2 \models F_2 \\
P_1 \overset{L}{\frown} P_2 \quad \Rightarrow \quad & \exists\, C_1 : S_1 \models C_1 \wedge \exists\, C_2 : S_2 \models C_2 \\
\wedge \quad & \left(F_1 \in C_1 \mid F_1 \models I_1 \wedge F_2 \in C_2 \mid F_2 \models I_2 \right) \mid F_1 \overset{L}{\rightharpoonup} F_2 \\
\wedge \quad & \exists\, T \mid \quad L \vdash T \wedge P_1 \vdash T \wedge P_2 \vdash T
\end{aligned}
$$

3.4 Deployment

Last, the third layer of **DAD** models first focuses on the definition of a deployment infrastructure based on *physical* types, as presented in Section 3.2.3. Second, mapping rules are specified between a concrete architecture configuration, *i.e.* Assemblage, onto the representation of the *running* infrastructure. Figure 3.7 spotlights on the **Deployment** modeling elements with the linked **Definition** and **Assemblage** constructs to highlight the relations between the three modeling layers.

We detail here the **Deployment** modeling elements and mapping rules. We afterwards specify the validation conditions used to ensure that the communication ports and paths provided by a deployment infrastructure are able to support the architecture **Assemblage**.

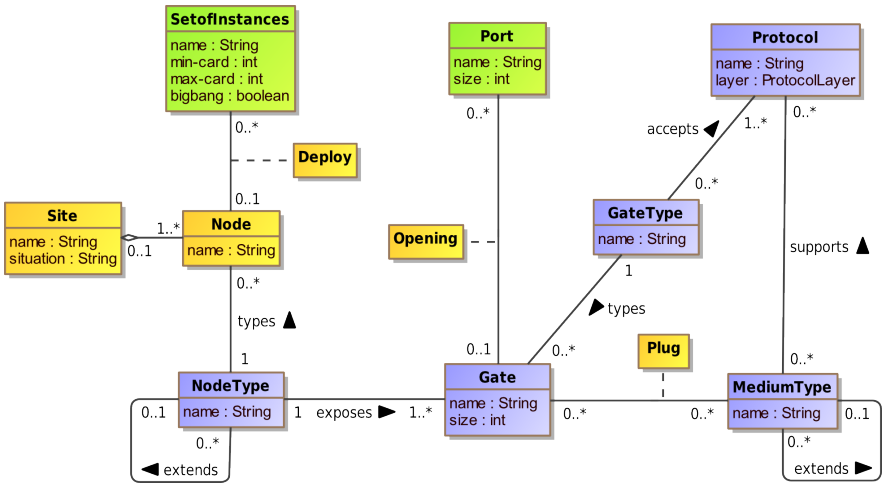


Figure 3.7: **Deployment** modeling elements

Node

A `Node` is an abstraction of a software environment or hardware that is part of a deployment infrastructure and may host one or more `SetOfInstances` that are deployed on it. It is typed by a `NodeType` and identified by a *name*. A `Node` inherits from the `Gates` exposed by its type. It is used to model an infrastructure of interconnected computation entities that will be used to support a particular `Assemblage`.

Site

A `Site` is a geographical place that contains `Nodes`. It is used to define a cluster or grouping of `Nodes` that are located at the same place. It is identified by a *name* and refined by a *situation* description. The `Site` is mainly used for documentation and visualization purposes. Further user-defined properties may be added to a `Site` definition, if needed.

Deploy

A `Deploy` is a mapping rule from a `SetOfInstances` to a `Node`. It depicts the abstract deployment of a software resource onto a running environment. Checks may be performed in case of compatible user-defined properties have been expressed on `SetOfInstances` and `Nodes` regarding available disk space, runtime environment, required system architecture, etc.

Plug

A `Plug` is the action of plugging a `MediumType` between two `Gates` in order to support the communication between `Nodes`. This rule is used to ensure that a `Gate` is actually connected to a communication path that supports its definition in terms

of `Protocol`. At this level, paths between `Nodes` will support the `Linkages` defined at the **Assemblage**, these `Linkages` using a particular `Protocol` too. Formally, let:

- N denotes a `NodeType`, being a set of `Gates`
- H denotes a `Node` and inherits from the `Gates` defined in its typing `GateType`
- G denotes a `GateType`
- A denotes a `Gate` and $\boxed{A \in H}$ denotes that A is exposed by H
- $\boxed{\models}$ denotes the *is typed by* relationship, such that $\boxed{H \models N}$ and $\boxed{A \models G}$
- M denotes a `MediumType`
- $\boxed{\vdash}$ is overloaded such that $\boxed{M \vdash T}$, T supports the Protocol T

If a `Plug` exists between two `Gates` with a `MediumType` M, noted by $\boxed{\overset{M}{\smile}}$, the following conditions must hold:

$$\begin{aligned} &\forall\, A_1 \in H_1 \quad \wedge \quad \forall\, A_2 \in H_2 \\ &A_1 \overset{M}{\smile} A_2 \quad \Rightarrow \quad \exists\, T \mid \quad A_1 \vdash T \wedge A_2 \vdash T \wedge M \vdash T \end{aligned}$$

`Opening`

An `Opening` rule expresses that a `Gate` is opened to receive calls for a given `Port`, *i.e.* a given `Port` is deployed on this `Gate` and makes available calls to the behavior depicted by the typing `Interface`. Many `Ports` may be accessible on the same `Gate`, but one `Port` cannot be reachable via multiple `Gates`. However, a `SetOfInstances` may have multiple `Ports` typed by the same `Facet` such that the same `Interface` may be exposed on multiple `Gates`.

The `Opening` rule ensures that, if there exists some `Linkages` defined in an **Assemblage** clause with `Ports` that are *opened* on the involved `Gates`, then they must support the `Protocol` used in the `Linkage`. Formally, let

- $\boxed{\hookrightarrow}$ denotes the `Deploy` rule, such that $\boxed{S \hookrightarrow H}$
- $\boxed{\circledcirc}$ denotes an `Opening`, such that $\boxed{P \circledcirc A}$, the `Port` P is opened on A

Then, the following conditions, must hold:

$$\begin{aligned} &\forall\, A_1 \in H_1 \quad \wedge \quad \forall\, A_2 \in H_2 \quad \wedge \quad \forall\, P_1 \circledcirc A_1 \quad \wedge \quad \forall\, P_2 \circledcirc A_2 \\ &P_1 \overset{L}{\frown} P_2 \quad \wedge \quad A_1 \overset{M}{\smile} A_2 \quad \Rightarrow \quad \exists\, S_1 \mid P_1 \in S_1 \quad \wedge \quad \exists\, S_2 \mid P_2 \in S_2 \\ &\qquad\qquad \wedge \quad S_1 \hookrightarrow H_1 \quad \wedge \quad S_2 \hookrightarrow H_2 \\ &\qquad\qquad \wedge \quad \exists\, T \mid \quad L \vdash T \quad \wedge \quad M \vdash T \\ &\qquad\qquad\qquad\qquad \wedge \quad A_1 \vdash T \quad \wedge \quad A_2 \vdash T \end{aligned}$$

3.5 User-defined Properties

In the previous Section, we briefly talked about user-defined properties. **DAD** model elements may be refined by *(key, values)* properties. The specification of such

properties is defined in dedicated libraries that can be imported into **DAD** models. They must be specified by the following attributes:

name an identifying property name
type the property type
target the modeling construct type to which such property can be assigned
order whether the type is (strictly) ordered (ascending or descending)
unit description and/or acronym of the measurement unit
semantics text-based description of the meaning of the property

Only the three first attributes are mandatory for a property. A property is always referred by its fully qualified *name*, so the name is identifying a property locally into its library. The *order* attribute is meant to be used with int, decimal and boolean property types only. For numeric values, an ascending order means that (strictly) higher values are *better* and a descending order means that (strictly) lower values are *better*. For boolean values, we defined the ascending order as $false$, $true$ and in the opposite way for a descending order. The available types are:

int a signed integer value (32 bits)
decimal a decimal value (32 bytes)
boolean a boolean value (`true` or `false`)
string a sequence of characters (limit of $2^{31}-1$ characters)
enum enumerated value (the enumerated type must be previously defined)

A property can be defined for a majority of **Definition** element types, *i.e.* `Services`, `Interfaces`, `ComponentTypes`, `LinkTypes`, `Protocols`, `NodeTypes`, `GateTypes` and `MediumTypes`, as well as for `Sites`. Also, *meta-information* regarding models may be added with the same mechanism, like authorship, versioning data, project management-related information regarding the development method, and so forth.

Properties for the same modeling element type may be grouped based on user-defined criteria. These correlated properties must then be specified in groups by the modeler. This is particularly useful to define boundaries of values, like a minimum and maximum connection bandwidth, for example. References to previously defined properties are also possible into a group. In that case, the sequence of the properties is important. For example, in the following code snippet 3.1, we define some properties and one group of properties that reuse another previously defined property[4].

```
1  package be.iodass.sample;
2  dadproperties group_sample {
3    // define the CPU frequency property
4    property for nodetype CPU {
5      type decimal asc; // ascending decimal values (higher values are better)
6      unit "Giga Hertz (GHz)"; // Unit of measurement
7      semantics "CPU core frequency in GHz"; // meaning of the property
8    }
9
10   // define an enumeration of values for the communication layer
11   enum CommunicationLayer {
12     physical, datalink, network, transport, session,
13     presentation, application, samespace
```

[4] all syntactic details will be further described in Section 3.7

```
14    }
15
16    // define the property for the communication layer
17    property for protocol layer {
18      type CommunicationLayer;
19      semantics "Communication layer: seven OSI layers with an extra one
20                 for processes sharing the same memory space";
21    }
22
23    // define the 'author' property
24    property for model author {
25      type string;
26      semantics "Model's author name.";
27    }
28
29    // define a group of properties that reuse the author previously defined
30    group for model {
31      modification_date {
32        type string;
33        semantics "Date on which a modification as been done into the model.";
34      }
35      author; // reference to previously defined 'author' property
36    }
37  }
```

Listing 3.1: Sample property definitions

First, a `CPU` property is specified as a `decimal` value that indicates the clock frequency of the *Central Processing Unit* expressed in *Giga Hertz*. An enumeration of values is also illustrated, based on the communication layer we discussed in Section 3.2.2. The enumeration is used to type a `layer` property for the `Protocol` modeling construct. The `author` property can be used individually or in combination with the `modification_date`, so wherever a modeler uses this property, he must also specify the `author`.

A *standard* language library has been defined with reusable properties and group of properties. We specified a list of `ProtocolAttributes` to add high-level properties regarding communication `Protocols`. Also, some meta-information can be added to any type of models such as authorship, creation date and versioning details. More details about the available libraries will be presented in Section 3.6.1 and 6.4.4.

3.6 Model Reusability

As pointed in Chapter 1 and 2, a needed asset for architectural languages nowadays is their *reusability* facilities. Two interrelated mechanisms have been developed to that purpose for **DAD** models: the possibility to define reusable libraries and to import these libraries into any **DAD** model. The present section presents both mechanisms.

3.6.1 Writing Reusable Libraries

The proposed language has been defined as flexible as possible. The modeling constructs we introduced in the present chapter, especially the infrastructure ones,

the data types and the communication `Protocols`, are intended to be refined and specialized by user-defined properties to create reusable building blocks. They can be placed in dedicated models for further reuse. Because many types of modeling constructs can be extended, as presented in Section 3.2.4, modelers have the possibility to reuse the definitions of previously defined construct and even rewrite part of their definitions in a flexible manner.

Also, architectural styles and patterns can be separately defined in **DAD Definition** models. Libraries of reusable solutions can be built that way and integrated into other architecture **Definitions** or instantiated into specific **Assemblages**. The separation between types and instances offered by both layers will be particularly useful to that purpose.

User-defined properties may also be placed in dedicated models to build domain-specific libraries. Combined to reusable **Definition** modeling elements, domain-specific architectural languages may be specified that way by modelers. Because **Definition** elements allow to model software as well as hardware components with many interconnection capabilities, a wide range of domains can be expressed on top of our modeling elements.

As we mentioned in the previous section, we wrote a standard library that comes together with the language. This library is composed by a set of `PrimitivesTypes` such as integer (`int`), logical value (`boolean`), floating point number (`float`), single character (`char`), character chain (`string`) and `byte`. A second part of the standard library (also referred as the *base* library) contains reusable properties that can be attached to modeling constructs.

3.6.2 Model Imports

As we will detail in Section 6.4.4, two mechanisms are available to import **DAD** library models into another model: implicitly or explicitly. The framework uses a naming convention to offer a location for implicit imports. By storing models in that specific place, any element defined in these models are implicitly imported into all new models[5].

We especially used that mechanism ourselves to create the aforementioned standard library with a list of primitive data types and meta-information regarding models, such as authorship, for example. We also defined some properties for infrastructure modeling elements to specify, for example, minimum requirements regarding `NodeType`'s CPU frequency and architecture, memory space, `MediumType` bandwidth or transmission rate and so forth.

At the opposite, explicit imports must be defined explicitly by dedicated statements in **DAD** models[6]. These models may then be defined anywhere and can be imported using their qualified names. An example of such import will be provided in the following section.

[5] this is valid for any type of models defined in the present dissertation, as depicted in the following two chapters regarding architecturally-significant requirements and model transformations.

[6] and this is also valid for the other types of models too.

Modelers are free to decide which strategy they prefer and both mechanisms may obviously be mixed, depending on user-defined rules or modeling habits for example.

3.7 The Online Book Library, a DAD Model Illustration

In order to illustrate the formalism presented in the current chapter, we will use a fictitious case study on an *online book library*. We first give an informal description of the case study. We then present the UML Use Case diagram [OMG, 2011d] that formalize the main functionalities for the involved subsystems. Finally, we build a set of **DAD** models to illustrate and discuss the main constructs.

The depicted example case study will be used thorough this dissertation to illustrate the other languages developed around the **DAD** formalism. This case study had also been used to validate our approach on a comparative case study we conducted to challenge it and gather feedbacks from users. The details concerning the case study itself will be presented in Chapter 7.

3.7.1 Case Study Overview

As an example, we will model a software system that allows customers to buy books on a web site. Customers will use their preferred web browser to connect to an online library webpage that offers a catalog of books. These books may be furnished by multiple bookstores, each of these stores being free to fix their own prices. The books being sold are then dispatched by a parcel deliverer directly from the bookstores to the customers.

Concretely, when a customer buys a book, the library organizes an auction between all available bookstores to determine the smallest actual cost and, by this mean, maximize its profit. The auction is lead by the library that contacts all bookstores, gathers all answers, selects the lowest price and contacts again all stores to request a new offer, lower than the previous price. This process is repeated until no store is able to lower its price for the given book. Once the winning store is found, the deliverer is requested to take care of the delivery from the store to the customer.

All stores must register to the library by sending their list of books with an indicative price. For each book, the stores have a cost price and a selling price (the indicative price sent to the library). They also freely define their own minimum margin, expressed as a ratio that is applied to the cost prices. This margin constraints the lower price limit they can propose for a book during an auction.

3.7.2 Main System Functionalities

We summarize the `Customer` functionalities of the three subsystems with a UML Use Case diagram in Figure 3.8.

The `Customer` has two main objectives on the `OnlineLibrary` subsystem (using the UML terminology). First, a `Customer` wants to consult a catalog of books. To this end, the `OnlineLibrary` must gather all catalogs from the `Bookstore` subsystems.

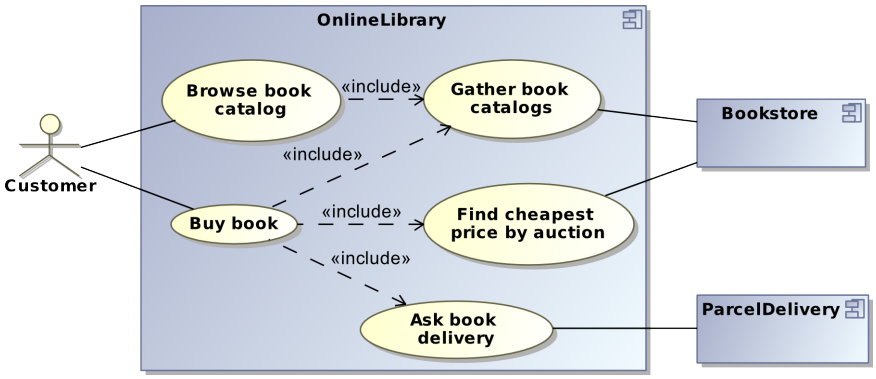


Figure 3.8: UML Use Case diagram of the `OnlineLibrary` subsystem

Second, the `Customer` wants to buy a book and this requires to know all `Bookstores` that have that particular book from the catalog, find the cheapest price via an auction and finally ask the delivery of the book to the `ParcelDelivery` subsystem.

3.7.3 A First Representation with a DAD Model

From these high-level requirements, we can specify a first naive architecture representation for the scoped system. This first model is simplified in a graphical representation of a **DAD** model in Figure 3.9. As a graphical formalism, we implemented some modeling elements in a UML profile[7] that extends the *component* and *deployment* diagrams.

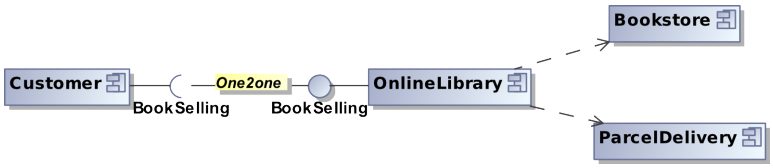


Figure 3.9: Simplified graphical representation of the `OnlineLibrary` **DAD** model

The `Customer` actor is represented as a `ComponentType` that requires a `BookSelling` `Interface` provided by the `OnlineLibrary`. Both `ComponentTypes` are bound with a `One2one` `LinkType`. The `OnlineLibrary` needs, *i.e.* through a `LeakUsage` dependency, the other two `ComponentTypes`, depicted by the dotted arrows. The complete model is illustrated and commented in Listing 3.2.

```
1 // a model always belongs to a package
2 package be.iodass.onlinelibrary;
3
4 // identifying name for the DAD model
```

[7] A discussion regarding the tool support is provided in Chapter 6, and more particularly in Section 6.4.4.

```
dadmodel onlinelibrary_naive {

  // definition layer
  definition {

    // book details (using primitive types defined in the standard library
    struct Book {
      int isbn;
      string title;
      string author;
      int year;
      string publisher;
      float price;
    }

    // interface in charge of the "Buy book" and "Browse book catalog" use cases
    interface BookSelling {
      sync Book[] browseCatalog(); // sync. operation that returns a list of books
      sync boolean buyBook(in int isbn); // sync. op. with input parameter
      event confirmDelivery(Book book); // event with Book parameter
    }

    // specification of a dummy Customer
    componenttype Customer{
      uses BookSelling as bs; // requires the service from the interface
    }

    // specification of the OnlineLibrary
    componenttype OnlineLibrary {
      implements BookSelling as bs; // provides the services
    }

    // empty specification of the stores
    componenttype Bookstore {}

    // empty specification of the deliverer
    componenttype ParcelDelivery {}

    // a basic one2one connector
    connectortype One2one {
      mode one2one; // point to point communication
    }

    // customer is connected to the library (from required to provided interface)
    linkagetype from Customer.bs to OnlineLibrary.bs with One2one;

    // the library needs the book catalogs and the auction service
    usage from OnlineLibrary to Bookstore;

    // the library transfers the customer and store details to the deliverer
    usage from OnlineLibrary to ParcelDelivery;
  }
}
```

Listing 3.2: Naive `OnlineLibrary` architectural representation

A model always belongs to a *package* and is identified by a *name* inside the *package*. The concatenation of the *package* and the *name* using dotted qualified names allows modelers to uniquely reference and import models. The **Definition** layer elements is grouped in a *definition* clause. The model is first composed by a `DataStructure` that defines a `Book` with some details. The `BookSelling` `Interface` is also defined with the `Services` that are proposed to the `Customer`. The `OnlineLibrary` and the `Customer` are connected through their `Facets` typed

by the `BookSelling` `Interface` with a `One2one` `ConnectorType`. The point-to-point connection is set up since the library is unique in the architecture, as defined in Section 3.7.1. The dependencies between the `OnlineLibrary` and the other `ComponentTypes` are not yet defined precisely, so usage dependencies are used instead of typed connections.

All user requirements are not already taken into account in this first model. We complement the first representation to add more details into the architecture, as graphically illustrated in Figure 3.10.

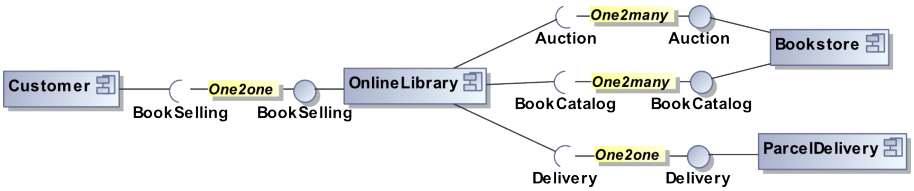


Figure 3.10: Graphical representation of the updated version of the `OnlineLibrary`

In Listing 3.3, we present the complete textual model with the new `Interfaces` and `LinkType`.

```
1  package be.iodass.onlinelibrary;
2  author : "Fabian Gilson"; // authorship user-defined property
3
4  dadmodel onlinelibrary {
5    definition {
6      struct Book { /* skipping details */  }
7
8      // customer details
9      struct CustomerDetails {
10       string name;
11       string surname;
12       string street;
13       int number;
14       string box;
15       int zip;
16       string city;
17       string country;
18     }
19
20     // bookstore details for the delivery of the book
21     struct BookstoreDetails { /* to be further defined later */ }
22
23     // details regarding the delivery
24     struct DeliveryDetails { /* to be further defined later */ }
25
26     // interface in charge of the "Buy book" and "Browse book catalog" use cases
27     interface BookSelling {
28       sync Book[] browseCatalog();
29       sync boolean buyBook(in int isbn);
30       event confirmDelivery(Book book, BookstoreDetails details);
31     }
32
33     // interface in charge of the "Gather book catalogs"
34     interface BookCatalog {
35       sync Book[] getBookCatalog();
36     }
37
38     // interface in charge of the auction process
```

```
39      interface Auction {
40        sync float getPriceForBook(in int isbn, in float currentprice);
41      }
42
43      // interface for the delivery
44      interface Delivery {
45        sync DeliveryDetails deliverBook(in Book book, in CustomerDetails customer, in
              BookstoreDetails store);
46      }
47
48      // specification of a dummy Customer
49      componenttype Customer{
50        uses BookSelling as bs;
51      }
52
53      // specification of the OnlineLibrary
54      componenttype OnlineLibrary {
55        implements BookSelling as bs; // provides the services to the customer
56        uses BookCatalog as bc; // requires the services to retrieve all catalogs
57        uses Auction as a; // requires the auction service
58        uses Delivery as d; // requires the delivery service
59      }
60
61      // specification of the stores
62      componenttype Bookstore {
63        implements BookCatalog as bc; // provides the catalogs of books
64        implements Auction as a; // provides the prices for the auction
65      }
66
67      // specification of the deliverer
68      componenttype ParcelDelivery {
69        implements Delivery as d; // provides the details for the delivery
70      }
71
72      // a basic one2many connector
73      connectortype One2many {
74        mode one2many;
75      }
76
77      // a basic one2one connector
78      connectortype One2one {
79        mode one2one;
80      }
81
82      // customer is connected to the library
83      linkagetype from Customer.bs to OnlineLibrary.bs with One2one;
84
85      // the library needs the book catalogs and the auction service
86      linkagetype from OnlineLibrary.bc to Bookstore.bc with One2many;
87      linkagetype from OnlineLibrary.a to Bookstore.a with One2many;
88
89      // the library transfer the customer and store details to the deliverer
90      linkagetype from OnlineLibrary.d to ParcelDelivery.d with One2one;
91    }
92  }
```

Listing 3.3: `OnlineLibrary` architectural style

Three more `Interfaces` are specified, that will be in charge respectively of the retrieval of the catalogs, the negotiation of the cost price for the books and the delivery to the customer. To that purpose, three more `DataStructures` are also identified, from which the `CustomerDetails` is specified. A new `ConnectionType` is defined to connect the library to the stores. Following the description given in the case study, there are many bookstores and a single library, so a `One2many`

`ConnectorType` is used to connect the `OnlineLibrary` to the `Bookstore`. Finally, the `ComponentTypes` are connected to each others through their `Facets` with this `One2many` `ConnectorType`.

The above model presents a type of architecture with three type of components interconnected through their compatible interfaces. Some details regarding the type of connection, point-to-point or one-to-many, are also expressed. However, the one-to-many connections should be further specified to refine the semantics of the connection, *i.e.* what kind of connection should be used: is the `OnlinLibrary` connected with multiple point-to-point connections to the `Bookstores` or in broadcast manner ? What are then the properties of the communication `Protocol` to use ? Moreover, how can we define a possible instantiation of this type of architecture ? In the next section, we will present how these pieces of information can be added into our model.

3.7.4 Adding More Properties and Specifying a Possible Instance

We depict in Figure 3.11 a possible instantiation of the architectural model presented in the previous section, with one `SetOfInstance` per `ComponentType`. These constructs are tagged with attributes to reference their types and to define their instantiation boundaries.

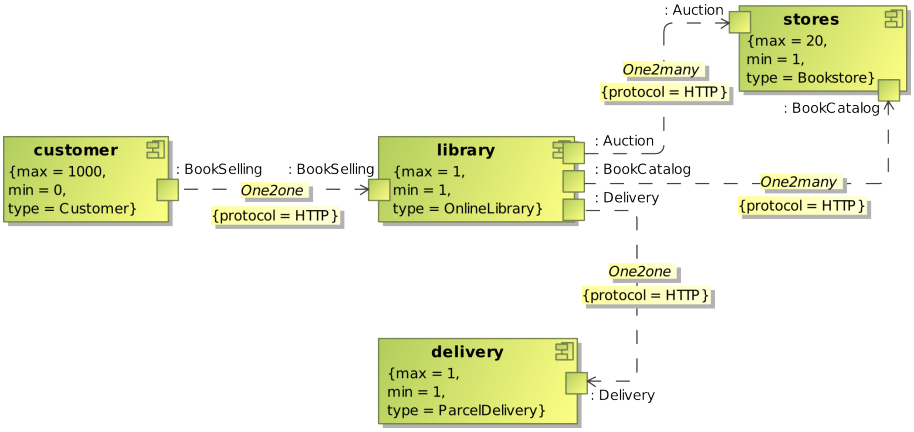


Figure 3.11: Graphical representation of an **Assemblage** for the `OnlineLibrary`

In order to refine the definition of the connections between the `ComponentTypes`, we introduce a `Protocol` into Listing 3.4 where we provide the code of the complete **DAD** model. We also refine the semantics of the `ConnectorTypes` and detail the **Assemblage** illustrated in Figure 3.11.

```
1 package be.iodass.onlinelibrary;
2 dadmodel onlinelibrary_upd {
3   definition {
4     struct CustomerDetails { /* skipping details */ }
5     struct Book { /* skipping details */ }
6     struct BookstoreDetails { /* to be further defined later */ }
```

```
7     struct DeliveryDetails {/* to be further defined later */ }
8
9     interface BookSelling { /* skipping details */ }
10    interface BookCatalog { /* skipping details */ }
11    interface Auction {/* skipping details */ }
12    interface Delivery {/* skipping details */ }
13
14    componenttype Customer{
15      uses BookSelling as bs;
16    }
17    componenttype OnlineLibrary {
18      implements BookSelling as bs;
19      uses BookCatalog as bc;
20      uses Auction as a;
21      uses Delivery as d;
22    }
23    componenttype Bookstore {
24      implements BookCatalog as bc;
25      implements Auction as a;
26    }
27    componenttype ParcelDelivery {
28      implements Delivery as d; // provides the details for the delivery
29    }
30
31    // new HTTP protocol
32    protocol HTTP {
33      layer application;
34      reliable : true;
35      ordered : true;
36    }
37    connectortype One2many {
38      mode one2many;
39      distributive : true; // means many point-to-point
40      accepts HTTP; // connectortype accepts the HTTP protocol
41    }
42    connectortype One2one {
43      mode one2one;
44      accepts HTTP; // connectortype accepts the HTTP protocol
45    }
46
47    // unchanged linkagetypes
48    linkagetype from Customer.bs to OnlineLibrary.bs with One2one;
49    linkagetype from OnlineLibrary.bc to Bookstore.bc with One2many;
50    linkagetype from OnlineLibrary.a to Bookstore.a with One2many;
51    linkagetype from OnlineLibrary.d to ParcelDelivery.d with One2one;
52  }
53
54  // let's define a possible instantiation
55  assemblage {
56    // the architecture may contain up to 1000 customer simultaneously
57    soi customer [0 1000] : Customer { Customer.bs as bs on HTTP; }
58    // the library is unique
59    soi library : OnlineLibrary bigbang {
60      OnlineLibrary.bs as bs on HTTP;
61      OnlineLibrary.bc as bc on HTTP;
62      OnlineLibrary.a as a on HTTP;
63      OnlineLibrary.d as d on HTTP;
64    }
65    // there could be up to 20 stores, but always at least one
66    soi stores [1 20] : Bookstore {
67      Bookstore.bc as bc on HTTP;
68      Bookstore.a as a on HTTP;
69    }
70    // the deliverer is unique
71    soi delivery : ParcelDelivery {
72      ParcelDelivery.d as d on  HTTP;
73    }
```

```
74
75      // linkages are almost a copy of the linkage types on instances
76
77      // up to 10 request can be handled at a time
78      linkage from customer.bs [0 10] to library.bs with One2one;
79      // library can be linked to 1 to 20 stores
80      linkage from library.bc to stores.bc [1 20] with One2many;
81      // library can be linked to 1 to 20 stores
82      linkage from library.a to stores.a [1 20] with One2many;
83      linkage from library.d to delivery.d with One2one;
84    }
85  }
```

Listing 3.4: Updated `OnlineLibrary` architectural model

The introduced HTTP `Protocol` is specified as a `reliable` and `ordered` communication protocol[8]. Both `ConnectorTypes` now accept this Protocol and `One2many` is defined as *distributive*.

In the **Assemblage** clause, the boundaries of the amount of possible instances for all `ComponentTypes` are specified. For example, we decided to fix the maximum amount of simultaneous `customer` instances to 1000. On the other side, there must always be at least one `stores` instance at every moment for the system to correctly function. For each `SetOfInstances`, the `Ports` that materialize the `Facets` of their typing `ComponentTypes` are exposed on the HTTP `Protocol`.

Finally, the `SetOfInstances` are connected following the patterns defined at the **Definition** layer. In concrete `Linkages`, modelers may specify constraints on the amount of simultaneous instances that can connect from a requiring instance to its providing counterpart. For example, the `library` can handle up to 10 requests from customers in a point-to-point connection, but at some point, no `customer` may be connected. At the opposite, the `library` must always know about at least one `stores` instance to be able to handle `customer` requests. The `library` is also tagged as *bigbang* to highlight the need to manually create it. The `customer` being a component out of our hands, its instantiation is out of our scope. The other two `SetOfInstances` could have also been tagged, but we decided here to represent the model from the `library` *point-of-view*, the other `SetOfInstances` being possibly managed by other instances.

From an **Assemblage**, we can now define a deployment mapping to an abstraction of a running platform. In the next section, we first specify some infrastructure constructs and then write mapping rules between the **Assemblage** and a running **Deployment** configuration.

3.7.5 Defining a Possible Deployment

We will now illustrate the **Deployment** layer in Figure 3.12, that will be further detailed in Listing 3.5. We present five groups of `Nodes`, bound by two different `MediumTypes`. The *host* attribute represents the hosted `SetOfInstances` by the

[8] these properties are also part of the standard library of DAD models. A reliable protocol ensures that all transmitted packets are received in the same order they've been sent by the sender. An ordered protocol ensures that all transmitted packets are received in the same order they've been sent by the sender.

`Node`. Thwo `Nodes`, namely the `gateway` and `server`, are also placed into a geographical `Site`, named `TheOffice`. The `gateway` does not host any `SetOfInstances`, but simply depicts a network gateway placed at the entrance of `TheOffice`.

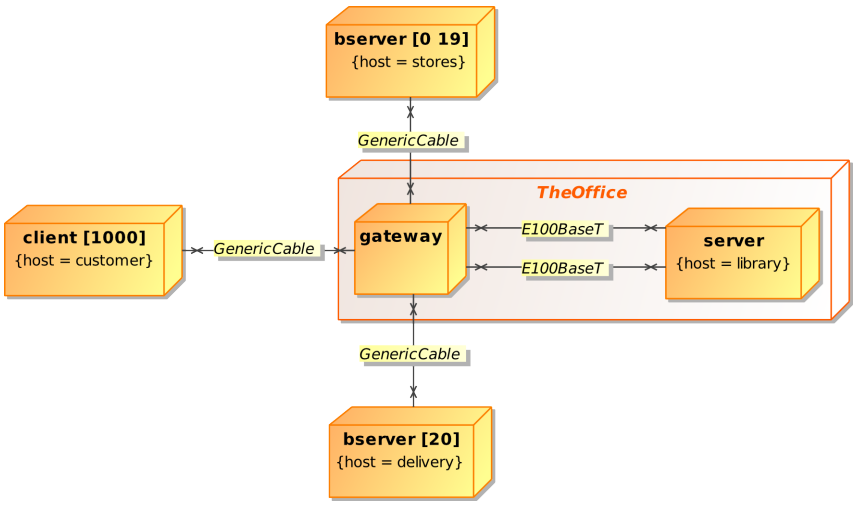


Figure 3.12: Graphical representation of the **Deployment** platform.

In Listing 3.5, we illustrate the extension mechanism for some types of constructs. The **Deployment** mapping rules are specified in the *deployment* clause. In the following listing, we also introduce the import mechanism to reference the model presented in Listing 3.4. Imported elements are referenced by their names as they were defined in the current model.

```
1  package be.iodass.onlinelibrary;
2
3  // models can be imported with their fully qualified names
4  import be.iodass.onlinelibrary.onlinelibrary_upd;
5
6  dadmodel onlinelibrary_dep {
7    definition {
8      // create a gatetype that accepts HTTP requests
9      gatetype Ethernet {   supports HTTP; }
10     // another gatetype that accepts HTTP requests too
11     gatetype GenericGatetype { supports HTTP; }
12
13     // simple client nodetype
14     nodetype Client { Ethernet eth0; }
15
16     // another node type (with the same definition)
17     nodetype BasicServer { Ethernet eth0; }
18
19     // generic gateway (network modem)
20     nodetype GenericGateway { Ethernet[2] eth; GenericGatetype gengate; }
21
22     // an extension of the basic server with a second interface and some properties
23     // these properties are part of the standard library
24     nodetype Server extends BasicServer {
25       Ethernet eth1;
26       CPU : 3.2; // cpu frequency in GHz
27       CPUArchitecture : "Intel Itanium"; // CPU architecture
```

```
28        CPULogicalCore : 8; // number of logical cores
29        RAMSize : 64; // size in Gigabyte
30      }
31
32      // generic mediumtype that supports the HTTP protocol
33      mediumtype GenericCable {
34        supports HTTP;
35        DataTransmission : 1.5; // transmission rate in Mbits/s
36      }
37      // common Ethernet network cable
38      mediumtype E100BaseT extends GenericCable {
39        Bandwidth : 100; // bandwidth in MHz
40        DataTransmission : 100;
41      }
42    }
43
44    deployment {
45      node bserver[21] : BasicServer; // create 21 nodes for stores and deliverer
46      node server : Server; // one server for the library behind a gateway
47      node gateway : GenericGateway; // library gateway
48      node client[1000] : Client; // client machines
49
50      // deploy all set of instances (indexes starts at 0)
51      deploy customer[0 999] on client[0 999];
52      deploy library on server[0];
53      deploy stores on bserver[0 19];
54      deploy delivery on server[20];
55
56      // geographical site that contains the server and the gateway
57      site TheOffice {
58        situation "Where our office is";
59        contains server, gateway;
60      }
61
62      // open the software interfaces (ports) on network interfaces (gates)
63      open library.bs on server::eth0; // customer requests on eth0
64      open library.bc on server::eth1; // book catalog requests on eth1
65      open library.a on server::eth1; // auction requests on eth1
66      open library.d on server::eth1; // delivery requests on eth1
67      open stores.bc on bserver[0 19]::eth0; // stores book catalog requests on eth0
68      open stores.a on server[0 19]::eth0; // stores auction on eth0
69      open delivery.d on server[20]::eth0; // delivery on eth0
70
71      // all clients are connected with a generic cable to the gateway
72      plug GenericCable from client[0 999]::eth0 to gateway::gengate;
73      // library server is connected with an Ethernet cable to the gateway
74      plug E100BaseT from server::eth0 to gateway::eth[0];
75      // library uses a second gate for its other ports
76      plug E100BaseT from server::eth1 to gateway::eth[1];
77      // bserver[0 21] are the nodes for the deliverer and the stores
78      plug GenericCable from gateway::gengate to  bserver[0 21]::eth0;
79    }
80  }
```

Listing 3.5: `OnlineLibrary` deployment constructs and mapping rules

In the above model, we introduce a `GenericGatetype` and an `Ethernet GateType` that can receive `HTTP` connections. The `Client` and `BasicServer NodeTypes` only declare an `Ethernet Gate`. A `GenericGateway` is defined with two `Ethernet` and one `GenericGatetype` gates, and represents a network gateway. The `Server` *extends* the `BasicServer` by adding a second `Ethernet Gate` and other user-defined properties, notably concerning the CPU. All these properties have been specified in the standard library we already referred to in Section 3.6.1.

Two MediumTypes are also defined, a GenericCable and its extension E100BaseT [IEEE, 2013]. The GenericCable supports the HTTP Protocol and guarantees a *TransmissionRate* of 1.5 Mbits/s. The E100BaseT overrides the *TransmissionRate* property and specifies a *Bandwidth* property of 100 MHz.

Now we have defined infrastructure constructs to model a target running platform, we can specify mapping rules between the concrete Assemblage, *i.e.* SetOfInstances and Ports, onto Nodes and Gates connected by MediumTypes. To this end, first, a list of Nodes are instantiated to receive the client, library, stores and delivery SetOfInstances (line 45 to 54 in Listing 3.5). The customer from index 0 to 999 are deployed on the Client Nodes with the same indexes. The same syntactic sugar is used to deploy the stores onto 20 BasicServer Nodes (from index 0 to 19) and the delivery on the remaining bserver. Basic checks are performed on cardinalities in the *deploy* clauses. The amount of deployed SetOfInstances must be sufficient for the specified amount of Nodes. If some SetOfInstances have been specified in the **Assemblage** and remains un-deployed, a warning message is issued.

Second, a geographical Site is declared and it contains the server (so the library software application) and the network gateway.

Third, the software Ports are made available from the outside on Gates (line 63 to 69). All Ports from the SetOfInstances depicted in Listing 3.4 are mapped to Gates. The library will use two different Gates to handle, on one hand, the client requests and, on the other hand, the communication with the other components. Like for *deploy* clauses, shortened statements can be used for *open* clauses, like on line 69 in Listing 3.5.

Last, we concretely connect Gates to each other to define a running configuration of interconnected Nodes. Communication paths are created between Gates to reflect the Linkages defined at the **Assemblage** layer. Analogously to the usage of LinkTypes in **Assemblage** connection clauses, we use directly MediumTypes in binding rules between Nodes because we focus on the properties attached to a logical and physical communication paths instead of the amount of logical connections or network cables we would need. For example, at line 72, by *plugging* a GenericCable from all clients to the gateway, we specify that all clients may join the gateway via its gengate Gate disregarding how the clients are *physically* connected to the gateway. These client Nodes are maybe placed behind a modem and multiple network routers, but these considerations are out of the scope if this particular model, even if they could have been modeled too.

3.7.6 Concluding Remarks on the Example Case Study

In the above example, we illustrated a significant part of the modeling constructs offered by the **DAD** language. However, we did not provide a complete example with all possible modeling elements, but we intended to exhibit the main assets of the proposed language. Other aspects of the **DAD** structural language will be provided in Chapters 6 and 7.

3.8 Wrap-Up and Conclusions over DAD Modeling

In the present section, we introduced the **Definition-Assemblage-Deployment** formalism, a component-based modeling language that integrates communication, runtime infrastructure and user-defined properties facilities. We advocated the usage of a layered representation for system architectures. We detailed the available modeling elements to define abstract architectures, concrete instance models with *dynamicity*-related properties, deployment mapping rules and custom properties. We also presented the binding mechanism used to connect compatible interfaces, close to the *duck typing* in object oriented programming. We finally illustrated this structural formalism with a case study on a book library system.

As a first objective, the formalism had to allow system designers to model software and hardware components in a lightweight and reusable way with a separation between *types* and *instances*. The layered representation of **DAD** models was designed to that purpose.

Likewise, hardware and deployment constraints play a major role in component-based system development since the amount of technological frameworks and running platforms exploded in the last years. User-defined properties may be easily specified for many types of modeling constructs in **DAD** models. These properties may refine the semantics of modeling elements and build libraries of reusable constructs or domain-specific architectural languages.

Besides, one of the main goals of the **DAD** modeling language was to define extensible communication facilities for component-based systems. Some effort has been put into the communication-related constructs and a taxonomy of linkage types between components has been defined. Communication protocols may be specified with user-defined properties and the separation with the connection support serves this flexibility to define architectural configurations. Furthermore, the linkage mechanism between interfaces based on the verification of services' signatures instead of type hierarchy reinforce system *evolvability* and composition flexibility.

CHAPTER 4

ARCHITECTURALLY SIGNIFICANT REQUIREMENT MODELING

The present chapter concentrates on the definition of architecturally-significant requirements and, on the one hand, the relations between them, and on the other hand, the relations to architectural constructs. We first define the notion of architecturally-significant requirements. We then recap the advantages and usages of design rationale. We introduce afterwards our ***Architecturally-Significant Requirement*** *(****ASR****) formalism and detail its modeling elements. Next, we suggest to adopt a semi-formalized method to write requirements and discuss the advantages of using such an approach in our context. Based on the case study depicted in the previous chapter, we finally illustrate the* ***ASR*** *modeling language.*

4.1 What are Architecturally Significant Requirements ?

In Section 1.3, we presented several design rationale and requirement traceability methods and tools. We identified their expected outcomes and assets in the context of system architecture design and knowledge management, mainly in terms of maintenance and knowledge transfer activities.

In the present dissertation, we focus on *architecturally significant requirements*, defined as requirements having *« a measurable impact on the software system's*

architecture » [Chen et al., 2013]. However, the derivation from business goals to such requirements is far from straightforward. There is often a « *mismatch* » between what information is expressed in a business goal and what an architect effectively needs [Clements and Bass, 2010]. Most requirements are often exclusively related to business goals or concentrate on expected functionalities, which are not always the best drivers for a system architecture. The link between both business and architectural goals (or quality requirements) has been mainly addressed by goal-oriented or template-based techniques [Gross and Yu, 2001; Velasquez and Weiss, 2006; Clements and Bass, 2010; Open Group, 2011]. They concentrate on keeping explicit this link, but still does not provide support between the architecturally significant requirements and the architecture model itself. Some of the approaches we summarized in Section 1.3 attempted to address this problem.

We observed to some extent a kind of consensus in the type of relations between requirements, design decisions and rationale. From the *ontology* proposed by Kruchten *et al.* [Kruchten et al., 2006] and the *core model* of de Boer *et al.* [de Boer et al., 2007] to the classification proposed by Zimmermann *et al.* [Zimmermann et al., 2009], design decisions and rationale have been raised as *first-class* entities in system development activities and documentation. As we mentioned earlier, recent experience reports, surveys and case studies highlighted some benefits and targeted several positive outcomes by integrating these concepts in system design methods and models.

In this chapter, as a first objective, we will focus on how architectural requirements can be recorded and attached to architecture model elements in a simple and non-intrusive manner. A second objective is to add rationale-based information to these requirements to trace the motivations behind the decision that created the architecture model, as well as the explored design alternatives.

4.2 Design Rationale, Is It Worth the Pain ?

But why do we really need to trace design rationale ? And what kind of rationale should be recorded ? As advocated in Section 1.3, especially in the experimentations conducted with the various techniques and tools dedicated to decisions and rationale traceability, these pieces of information play a key role in software maintenance and architectural knowledge transfer.

More and more researchers and practitioners are convinced by the utility of such methods for the software engineering field [Gruber et al., 1991; Carroll, 1996; Tang et al., 2006; Ali Babar and Lago, 2009; Lago et al., 2010]. However, probably one of the main obstacles to their adoption is the extra amount of work needed to encode the decision-making process with its justifications.

Instead of settling the matter, we suggest to let the decision in the project managers and designers' hand. Every project may have its own rules, standards and expectations regarding the completeness and quality of the deliverables. Even, every decision regarding the architectural design may not be critical enough to require an exhaustive and formalized argumentation. According to empirical observations [Ramesh and Jarke, 2001] and survey [Malavolta et al., 2013], the link between

architectural artifacts and requirements should be kept in a non-obtrusive manner with, at least, some details on the reason why the requirement is fulfilled by the implementing element. Mechanisms should also be provided to relate requirements to each other in order to help designers to make trade-offs between alternative solutions or understand the consequences of selecting a particular solution regarding the other impacted requirements.

In the following section, we introduce a set of modeling and recording facilities for architecturally significant requirements and design rationale. Even if the proposed modeling constructs are related to the **DAD** structural elements presented in the previous chapter, the concepts and relations between requirements can be easily reused in combination with other *component-and-interface* languages.

4.3 Architecturally Significant Requirement Modeling

We present now the concepts and relations of our formalism, simply called **Architecturally Significant Requirement** (**ASR**). We first reproduce the meta-model of the **ASR** language. Afterwards, we discuss all introduced concepts in two categories, first the requirement-related modeling elements and relations, and second, the types of rationale that can be recorded into an **ASR** model.

4.3.1 Requirement Language Specification

Our meta-model is highly inspired from the ontology of Kruchten *et al.* [Krutchen, 2004; Kruchten et al., 2006] and the classification of Zimmermann *et al.* [Zimmermann et al., 2007, 2009]. We reused many concepts from both approaches and, for some of them, slightly adapted their definitions to stick to our domain or intents. Figure 4.1 depicts the **ASR** meta-model.

The language is separated into two main parts: the requirements with their possible relationships depicted in red and the type of rationale, highlighted in yellow. The `DADElement` is an abstract *supertype* for all **DAD** modeling elements we discussed in Chapter 3. The pink box that represents the `TransformationRule` will be detailed in Chapter 5. It consists in any transformation that can be applied onto a **DAD** model.

4.3.2 Requirement-related modeling elements and relations

The essential element in an **ASR** model is the requirement itself. In such a model, a set of relationships are available to bind either requirements to each others, to `DADElement` or to `TransformationRules`.

Asr

An `Asr` represents a requirement of the system that may influence the structure of the architecture model in terms of its topology, functionalities, qualities, and instantiation or deployment constraints. It is identified by a *name* and is either *functional* or *non-functional*, as specified by its *type* attribute. Its full definition is

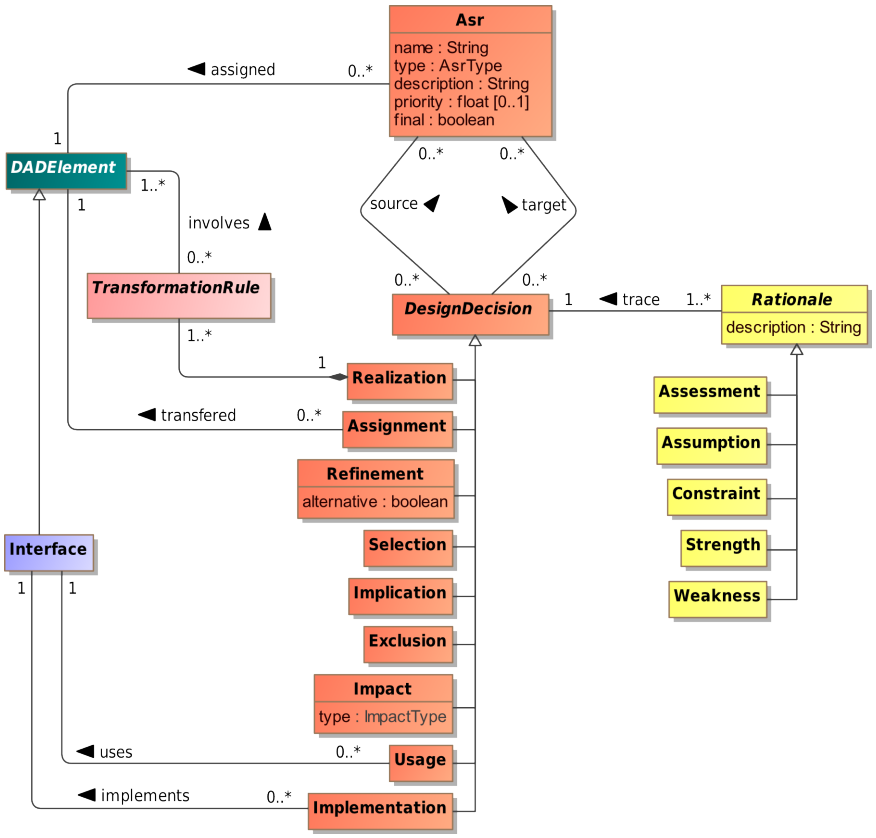


Figure 4.1: **Architecturally Significant Requirement** meta-model

recorded in the *description* attribute. A *priority* can also be assigned to an `Asr`. This *priority* is expressed with a float value, but the rules or prioritization method are left to the users. One may use it to represent a (strict) order of `Asr`'s implementations, or to specify a weight that will be used in an Agile planning poker [Grenning, 2002], for example. An `Asr` can also be flagged as *final* to avoid any other `Refinement` or `assignment` of it. This may be useful when the `Asr` expresses a concrete implementation solution from which no possible alternative may be explored, or to ensure the responsibility of its fulfillment may not be transfered to other model element, for example. It must be assigned to one `DADElement`, this model element being responsible of its fulfillment into the architecture model. The `DADElement` is then referenced as the *owner* of the `Asr`. Finally, it can be the source or target of one or more `DesignDecisions`.

DesignDecision

A `DesignDecision` is an abstract modeling element that represents any decision that can be taken onto an `Asr`. It always binds two or more `Asr` and it is *traced, i.e.*

justified, by one or more Rationale that document the link itself.

Assignment

assignment is the decision of committing the responsibility of an Asr to a DADElement. During development, as the analysis is conducted, some requirements may have been assigned to the wrong **DAD** model element or must be reassigned to finer-grained components. Also, as a system may evolve over time, the duty of some requirements can be transfered from one element to another, for example because of the usage of a dedicated technological framework or *component off the shelf*. Such assignment flexibility is thus needed, especially in iterative software development methods.

Refinement

Requirements may be split into one or more other requirements for multiple reasons, all expressed by a Refinement relationship. First, an incomplete or ill-defined requirement definition can be refined into one or more precise ones. Second, a broad requirement may be split into a set of *lower-level* ones, each of them having disjoint concerns. Third, alternative solutions may be expressed with the flag *alternative* set to *true*. If an higher-order Asr has multiple refining requirements marked as *alternatives*, they all provide a solution for the same concern such that only one of them should be selected at a time. In case an Asr is refined with requirements that are alternatives and others are not, all *alternatives* express substitutable solutions for the same matter. In other words, an Asr cannot be refined in multiple *group of* alternatives. If such a situation would occur, intermediate Asr must be created that will be refined into alternative Refinements.

By default, a lower-level requirement is assigned to the same modeling element than his higher-order parent. However, a refined Asr may be assigned to another model element. This situation typically happens when a fuzzily described Asr is split into more detailed pieces that must be implemented by distinct components, like, for example, when a system must gather some information that is provided by multiple external source systems. The ASR formalism is flexible enough to directly reassign sub-requirements to the right target model element that will be in charge to its fulfillment, such that no extra assignment decision must be taken separately.

Selection

Within an *alternative group*, at some point of the design process, an *alternative* Refinement will be selected. The Selection expresses the decision of choosing one *alternative* as the current implementation solution within all other *alternatives*.

Implication

An Implication between two Asr indicates that one requirement triggers one or more other requirements. Implications are typically specified on Asr that are

part of *alternatives* to define dependencies between a possible solution and its implications in other alternative groups if the `Asr` was selected. An `Implication` is unidirectional, so that in case of mutual implications, relations must be defined for all involved requirements in both ways.

Exclusion

An `Exclusion` is somehow the opposite relation of an `Implication`. It specifies a mutual rejection between one source `Asr` and a list of target `Asr`. It is used to specify a negative implication between requirements typically on alternative `Asr`. In this case, if an *alternative* requirement is selected, it will forbid the selection of other solutions in another group of alternatives. Unlike an `Implication`, the `Exclusion` relationship is bidirectional.

Impact

An `Impact` is any other type of relation between two or more `Asr`. It means that a requirement has some effect on other ones, the exact nature of the influence being project- or method-specific. An `Impact` is unidirectional and must be specified as having a *negative*, *positive* or *neutral* influence. The *neutral* type of `Impact` indicates that there is a dependency between the requirements, but the precise significance and/or consequences are unknown.

Usage

A `Usage` means that an `Interface` is required to fulfill the requirement. The `Interface` provides the needed behavior and properties that are required by the `Asr`'s owner. The `Interface` must either exist in the **DAD** model referenced by the current **ASR** model or it must be explicitly *imported*, analogously to the import mechanism for DAD models discussed in Section 3.6.2. A `Usage` dependency may be related to only one `Interface`. If an `Asr` requires more than one `Interface` to be fulfilled, it must be split, *i.e.* refined, into lower order requirements beforehand. `Usage` relationships, like `Implementations`, may only be used with `ComponentTypes`.

Implementation

An `Implementation` means that an `Interface` will be provided to fulfill the requirement. The `Interface` provides the expected behavior, such that the `Asr`'s owning `ComponentType` will implement it. The `Interface` must already exist in the referenced **DAD** model or must be explicitly *imported*. An `Implementation` dependency may relate one `Interface`, or similarly to `Usages`, the source `Asr` must be split into multiple refinements.

Realization

A `Realization` gathers more complex structural changes that must be applied to the referenced **DAD** model in order to fulfill the `Asr`. It will induce one or more

TransformationRules involving one or more DADElements. In Chapter 5, we will introduce a dedicated language to express transformation on **DAD** models, but transformations can be expressed in any transformation language, according to modelers or project management decisions.

4.3.3 Rationale-related modeling elements

Our requirement formalism is enhanced with the possibility to document the aforementioned type of decisions with design rationale. As depicted in Figure 4.1, every DesignDecision must be documented by a Rationale. Figure 4.2 recaps the available types of rationale for an **ASR** model, that are detailed right after.

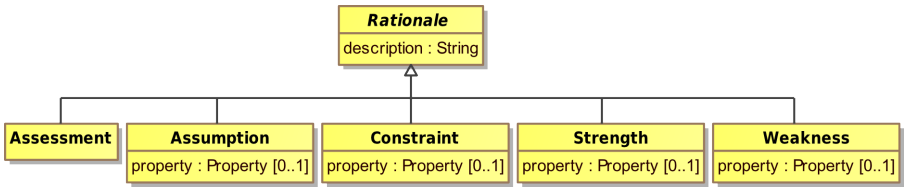


Figure 4.2: **ASR**'s rationale modeling elements

Rationale

A Rationale is an abstract element that represents any motivation or hypothesis behind a DesignDecision. It is usually composed by a text *description* where requirement engineers or modelers further explain their thoughts regarding their decisions.

Assessment

An Assessment is the only mandatory piece of information for a DesignDecison. It is used to justify the decision with the main reason why the decision has been taken. Alternatively or in case of template-based requirements (see Section 4.5.1), it may be used to reformulate or give more *unstructured* information regarding the requirement, especially when higher-level requirements are refined into lower-level and more precise ones. The completeness of the explanation is out of the scope of the present language. We simply offer a structured way of recording the major argumentation point hold by the modeler or to further detail the goal of the requirement.

Assumption

An Assumption represents any hypothesis taken on the system to build or the environment that can influence the decision. By definition, an Assumption is always considered as a true fact. As the other types of Rationale except the Assessment, an optional Property, as introduced in Section 3.5, may be added to the description

in order to formalize it in a structured way. This feature opens the possibility to verify if some hypothetic properties are not violated in a **DAD** model, for example.

Constraint

A `Constraint` expresses either a condition that must be taken into account regarding the decision and the `Assessment` of it, or a future consequence induced by the decision. Any type of constraint can be defined on the system to build, on the environment or on human resources, for example. A structured `Property` may be included in the *description* to possibly enable checks on a required `Constraint` for a given `Asr` and its existence in a referenced **DAD** model.

Strength

A `Strength` is used to support a decision with its possible advantages. One of the objective of this type of `Rationale` is for argumentation purposes when evaluating multiple alternatives. One may detail the benefits of one decision regarding an alternative design decision or even structural configuration, if the `Rationale` is bound to a `Realization` decision, for example.

`Strengths` are especially valuable to understand the viewpoint of the modeler in terms of expected qualities regarding the `Asr`. When considering multiple alternatives, modelers may evaluate the possible effects of a decision regarding some (non-functional) expectations on the system. With `Strengths`, they are able to specify clearly why a decision should be taken or why they select this particular alternative.

Weakness

A `Weakness` has the same purpose as a `Strength`, but has the opposite meaning. It records the shortcomings linked to an (alternative) `Asr`, possible limitations or known failures of an `Asr`. Similarly to `Strengths`, identified `Weaknesses` would probably focus on the expected qualities of the system under development or on the impact on its environment.

Both `Strengths` and `Weaknesses` are particularly valuable for architectural knowledge transfer or recovery because they record the identified aspects of a decision that were taken into account when evaluating the design alternative or decision. They further justify the criteria, *i.e.* viewpoint, of the modeler when (s)he took a decision regarding an `Asr`.

4.4 User-Defined Properties in ASR Model Too

Exactly as **DAD** models, some modeling elements from **ASR** models may be refined by user-defined properties, using the same mechanism as described in Section 3.5. These properties are first used to specify meta-information regarding the model or the requirements themselves. Ownership, timing or project specific information can then be added to these elements.

Second, *typed*-`Rationale` can be defined along with their descriptions. By offering a structured mechanism to define design rationale, the proposed language may support analysis of **DAD** models regarding the properties expressed in an **ASR** model and their effective fulfillment in the related **DAD** model. Remember an `Asr` is always assigned to a `DADElement`, depending on the type of `Rationale`, some consistency checks can be performed. Before going into details of these possible strategies and to be able to concretely illustrate the present discussion, we first introduce the **ASR** model formalism in its graphical representation in Figure 4.3 an with its full textual description in Listing 4.1. Then, we detail, per `Rationale`, the possible verification strategies that can be applied.

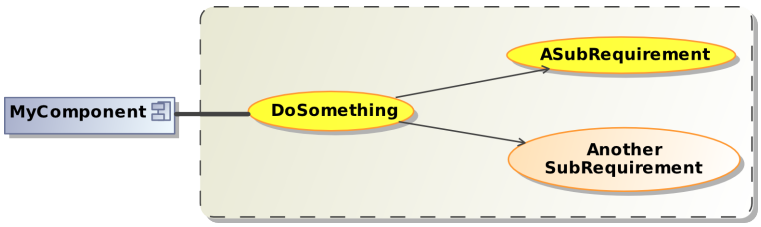


Figure 4.3: **Architecturally Significant Requirement** graphical sample

The graphical syntax is close to the UML use case diagram[OMG, 2011d] with ellipses for requirements and a box around all requirements that concerns a **DAD** model element. In this sample **ASR** model, we show three requirements that are assigned to the `MyComponent` `ComponentType`. The *"DoSomething"* functional requirement (depicted in a yellow ellipse) is refined into two lower-order requirements, from which the *"AnotherSubRequirement"* is non-functional (orange ellipse). The graphical syntax concentrates on the relations between `Asr`. As we will show in Section 4.6, the other types of relationships can be represented with dedicated graphical items. The detailed and commented textual model is presented in Listing 4.1.

```
1  package be.iodass.sample; // belong to a package, as DAD models
2  import be.iodass.sample.propsample; // import dadproperties model
3  asrmodel asrsample with be.iodass.sample.dadsample { // corresponding DAD model
4    // functional requirement with ID "DoSomething"
5    func DoSomething assigned MyComponent {
6      description "This requirement concerns a feature we absolutely need";
7      priority 1; // ordered priority
8    }
9    // functional requirement that refines DoSomething
10   func ASubRequirement assigned MyComponent {
11     description "The important functional feature";
12     priority 1;
13     refines DoSomething { // link to higher-level requirement with rationale
14       assessment "Concentrate on the functional aspect"; // mandatory
15       // assumption is defined as a structured property
16       assumption stateless : true; "The component type is stateless.";
17     }
18   }
19   // non functional sub-requirement that also refines DoSomething
20   nonfunc AnotherSubRequirement assigned MyComponent {
21     description "The important feature needs fast answers";
22     priority 2;
23     refines DoSomething {
```

```
24        assessment "Split functional and non-functional aspects";
25        // constraint can be formalized as a structured property
26        constraint responsetime : 10; "The component answers to requests in max 10msec";
27      }
28    }
29  }
```

Listing 4.1: **ASR** sample model

As illustrated in the above Listing, an `ASR` model is associated to a structural `DAD` model since each `Asr` must be assigned to a modeling object that will be the owner of the requirement, until no other `assignment` is performed. We can identify one main requirement `DoSomething` that is refined into two sub-requirements, a functional and a non-functional one. For both sub-requirements, the `Rationale` is composed by a user-defined property and a description, assuming that both properties are defined for `ComponentTypes` in the referenced **DAD** properties model. As its name suggests, the `stateless` property indicates that a `ComponentType` can be stateless, so that each call to its implemented `Services` are independent. The `responsetime` property defines the maximum response time in millisecond that the component need to answer to client requests. This simple mechanism that mixes user-defined properties and text-based descriptions may enable *automatable* verifications across models, as we will depict in the subsequent sections.

4.4.1 Active Asr and Property Verification

In order to define the following verification rules, we first have to introduce the notion of *active* `Asr`. An active `Asr` is a requirement that either is not part of an alternative group, or is a selected alternative in the current **ASR** model.

In order to perform analysis on the values assigned to `Rationale` with structured properties, we also define the notion of *incompatible* values. Verification on the compatibility may be performed on numeric, i.e. int and decimal values, and on boolean properties. The verification rely on the *order* attribute. If the order is (strictly) *ascending*, then any value (strictly) under the value of the current property is incompatible. For (strictly) descending types, any value (strictly) above the focused value is incompatible[1]. If no *order* attribute is specified, any different value for the same property will be considered as incompatible.

Both these notions are particularly important for the consistency verification of properties defined inside types of `Rationale`. Note that the following verifications can only be performed on boolean and ordered numeric properties. Also, the following strategies always use the type of `Rationale` under focus as the current point of view. Conflicting verifications can arise if strategies are mixed and, usually, manual trade-offs or priority between the strategies are required.

[1] Remember that we specified the order attribute for boolean values in Section 3.5 too, with ascending values defined as the order $false - true$ and descending as $true, false$.

4.4.2 Assumptions

Since an Assumption is an hypothesis considered to be always valid, for any user-defined property expressed as an Assumption in an **ASR** model, no counter example may be found for the owner of the requirement in the referenced DAD model. For example, if the modeler determined that the responsetime of a ComponentType must stay beyond a certain limit, the owner cannot be tagged with the same property and a worst value. Or, like in the example above, if the ComponentType is intended to be *stateless* and this property as no defined *order*, it cannot be tagged as $stateless : false$ in the corresponding **DAD** model for the same ComponentType.

In general, all properties defined as Assumptions in an Asr definition for a given DADELement cannot be specified with an incompatible value in the referenced **DAD** model. For *unordered* properties, the value must be the same in the Rationale property and in the definition of the owner DADElement.

Similarly, consistency checks can be performed for all Assumptions across all *active* Asr assigned to the same DADElement to ensure no incompatible hypothesis are made on the same modeling element. If such incompatibility is found, a manual trade-off must be found between all values to determine which one is actually the most *plausible* or *acceptable* for the requirement engineers or system architects.

Also, no Constraint in an active Asr can be more *stringent* that an Assumption for the same element. What would it mean if, for example, a component's *responsetime* is estimated to stay under $10msec$ in an Assumption and an Asr requires it to be under $5msec$ in a Constraint ? Here again, some manual decision must be taken to decide which value must be used for that property.

The validation is more straightforward regarding Strengths and Weaknesses because the latter are used for documentation purpose. No incompatible values can be found in an active Asr, or they should be simply replaced by the value specified in the Assumptions since they are used for argumentation purposes.

4.4.3 Constraints

Constraints express properties that must be respected by a modeling element. Assumptions with incompatible values in active Asr should be avoided. For example, if a component is constrained by a $responsetime : 10msec$, no Assumption can go beyond this limit, since the responsetime is defined as an ascending integer value.

Also, compatibility checks can be performed in the referenced **DAD** model to ensure that the Constraint property is not violated in the **DAD** model, *i.e.* no incompatible values are found for the same modeling element.

4.4.4 Strengths and Weaknesses

Strengths and Weaknesses are devoted to influence and support the design decisions, so they are mainly used as documentation means. For any property defined as a Strength or as a Weakness, it may not be incompatible with an Assumption or a Constraint assigned to the same DADElement.

4.5 Clues for The Right Level of Details

In our **ASR** models, we decided to put forward semi-formal requirement definitions. Text-based recording in requirement engineering tools is still very popular in the industry [Liu et al., 2010; Carrillo de Gea et al., 2012]. Requirement documents are still often made of plain text in natural language, sometimes illustrated with some UML diagrams or ad-hoc dialects. Many companies also rely on their own template documents that can change from projects to projects, depending on the team manager or even external consulting companies with their own standards.

However, the current practice is moving to more formal techniques for requirement recordings, system design and verification [Woodcock et al., 2009]. We argue here that a combination of semi-formalized requirement specification based on structural *templates* coupled with *formal relationships* between requirements and *design rationale* can improve the requirement exploration as well as the architectural knowledge of the system to build. Some recent experiments or usage reports of explicit recordings of rationale tend to highlight the aforementioned benefits in controlled environments or in the industry [Bracewell et al., 2009; van Heesch et al., 2013].

4.5.1 Template-Based Requirements

Many document templates have been proposed to write requirements in a structured and referable manner. *The Open Group Architecture Framework* (TOGAF) we mentioned in Chapter 2 provides a list of text documents with predefined sections for all phases addressed by the TOGAF® framework [Open Group, 2011]. For all documents, versioning and authorship information must be provided. In its *Architecture Requirements Specification* document, the framework requires modelers to provide details regarding the *architecture requirements*, the *interoperability requirements*, the *constraints*, *assumptions* and *success measures*. Except regarding this last piece of information[2], all other sections may be written in a structured way in our framework.

The *Volere* requirement specification template, from the *Atlantic Systems Guild Limited* is another example of a widely used template [Robertson and Robertson, 2012]. This document is articulated around five categories: *project drivers* (purpose and stakeholders), *project constraints* (naming conventions and assumptions), *functional requirements*, *non-functional requirements* and *project issues* (open issues, risks, costs and documentation). It reuses UML diagrams [OMG, 2011d] to depict functional use cases but requirement engineers may simply use a requirement table which is very similar to a simple spreadsheet. For behavioral or data specification, no prerequisite or suggested formalism is specified. Only a UML class diagram is presented as an example for a *data* representation. However, as the TOGAF templates, it does not provide structured referencing mechanisms between requirements so that refinement or conflicts between them can only be expressed textually.

[2] Even if such a user-defined property could be easily defined in a **DAD** property model, like the versioning and authorship properties from the built-in library.

Mavin *et al.* propose the EARS templates to write requirements in a structured way [Mavin and Wilkinson, 2010; Mavin, 2012]. Two types of requirements can be expressed: *normal operations* (expected applications) or *unwanted behaviors* (exceptions handling). We extended the general EARS template to specify the counterpart system in the communication, if any. All requirements must be written around this generic **xEARS** template, of the form:

> <*pre-condition(s)*> <*trigger*> **the** *system* **shall** *response* <*counterpart(s)*>

Optional parts are denoted between angle brackets ("<>"). Every part must always be put at the same place, with the following meaning:

pre-condition prerequisite to the requirement (zero to many)
trigger discrete event that causes the requirement (one at most)
system identifiable and explicit name
response behavioral reaction of the system (one to many)
counterpart referable system involved in the communication (zero to many)

Simplest requirements are called *ubiquitous* and are defined as the most simple form of the template without any optional parts. They consist in requirement of the form « ***the system shall*** *do something* ». The authors defined a set of four typical constructions as pre-conditions or triggers:

state-driven (*while*) behavior active while some condition stands
event-driven (*when*) condition triggered at the system boundary
option (*where*) behavior activated if presence of an optional component
unwanted (*if*) answer to an undesired event or failure

Our custom *counterpart* extension allows to define the communication between the system that is the subject of the requirement and the other involved systems. Two types of dependencies can be expressed, the required dependencies and the provided ones. Both constructions are useful to depict dependencies between systems, especially for higher-level requirements that may have a broader scope and describe more complex behaviors or system qualities. They are identified by the keywords *from* and *to*.

required (*from*) system that provides (part of) the requirement
provided (*to*) system that requires the requirement

All these constructions can also be combined to create complex requirement definitions. The EARS templates have been developed and used in the aeronautic industry and they are becoming more popular also in IT companies [Terzakis, 2013]. Concrete examples of the usage of the extended EARS templates will be provided in Section 4.6.

When using Volere or TOGAF-like templates, no structured strategy for referencing, decomposing or defining requirements is suggested. Requirements may be written at any level of details. These document templates offer *boxes* to be filled with some information, but let free the requirement engineers to write, split or group requirements according to their own method. At first sight, this practice looks fair

enough, but we believe that, at least, a structured mechanism should be provided to record design rationale and relations between requirements.

Approaches like the EARS templates, with predefined slots and keywords in *pattern* sentences are definitely promising since they open the door for text extraction possibilities in order to migrate or transform textual requirements to formal models. With such a writing strategy, requirement engineers may stick to their habits if they were used to textual descriptions, but with appropriate tooling support, some verifications may be applied on extracted requirements models.

However, such alike templates by themselves miss the capability to easily cross over requirements and alternatives without going into the textual description itself. No structured relationships may be defined between requirements which reduce the ability to identify dependable or conflicting solutions, for example. Therefore, in the next section, we discuss the possible advantages of using formal relations between requirements.

4.5.2 Formal Relationships

As discussed in Section 1.3, many researchers advocated for structured relations between requirements as an important feature in requirement engineering. Moreover, as pointed by practitioners in the aforementioned survey on architectural languages [Malavolta et al., 2013], the link between system requirements and architectural artifacts is crucial, especially to lower the cost of architectural changes.

As discussed in Section 1.3.10, many modeling languages uses formal relationships between business goals and/or requirements. They usually define formal decompositions of higher-order goals into lower order ones or to represent strategic decision-making processes with involved stakeholders. In some sense, we share a common view of formalized relationships, where the *actors* are system elements, but the only goals we are interested in are the ones having an influence on the system's architecture.

The **ASR** formalism focuses on two types of relationships: between requirements themselves and with architectural constructs. The main goal of the first type of relations is to help requirement engineers, as well as system architects, to identify refinements, alternatives and any other type of conflicts between architectural requirements. These pieces of information are especially valuable during design phases to help with decisions, highlight possible problems induced by these decisions and trace the decomposition of higher-level requirements into lower-level ones. The second type of relations, with architectural constructs, is particularly useful for maintenance and knowledge transfer activities. A particular requirement, especially the ones that are translated into architectural aspects or patterns, is often diluted into the overall model. The roles played by every construct can then become unclear and the system concrete architecture starts to *derive* from its representation. This phenomenon has been identified during the early ages of the software architecture discipline as the *architectural drift* and *erosion* [Perry and Wolf, 1992] we already discussed in Section 1.2.13. Both phenomena contribute to increase the

architectural rigidity to structural changes and can be partially tackled by reinforcing the link between the requirements and their implementation artifacts.

But in order to further enhance change traceability and architectural knowledge documentation, the rationale concerning *principal* design decisions should also be recorded along with the requirements and architecture models. Furthermore, this recording activity should be done in a straightforward and *tool-supported* manner to lower the extra work needed and lower the cost for architectural documentation activities.

At the opposite to many graphical approaches, in our requirement modeling language, the links between `Asr` must be defined *the other way around*, from the target to the source. This may seen quite counter-intuitive, but we designed it this way to avoid having to crawl all over a requirement listing every time we add an alternative or any other type of relationship. For example, the `Refinement` link is specified in the `Asr` that refines a higher-order one. Similarly, new alternatives may be added into a listing without modifying the content of the `Asr` to which the alternatives points to.

4.5.3 Rationale or Not ?

This question could be reformulated into "*when does it become ineffective to record a design rationale ?*" We believe there is no such a universal answer to this controversy, but this is instead a matter of project management and development context. From a project to another, the prescriptions regarding documentation may be low or high, based on, for example, company policy, expected time-to-market, development method, available tool support, etc. Also, depending on the type of system to be developed, the level or decision traceability may vary too. For example, the documentation expectations are different for a non-critical system that will be thrown away in a couple of years than an autopilot system for a plane.

With **ASR** models, requirement engineers and modelers are free to add as much information as they want. The mechanisms introduced in the preceding sections are flexible enough to be adapted to a wide range of project-specific managements. The only mandatory rationale is the `Assessment` description that expresses the main reason behind the decision taken.

One of the most important feature for an architectural documentation formalism is probably its simplicity and lightness. As highlighted in Section 1.3, the perceived *return on investment* of architectural documentation frameworks is usually low, even if many practitioners admit its usefulness. By recording the design rationale directly together with the requirements in a free-text format, the additional work is limited, especially because only few details are mandatory. Relations between requirements are also expressed in a simple manner, without having to draw additional models.

4.5.4 A Hybrid Visualization for a Hybrid Definition

To fully benefit from our hybrid requirement definition mixing text-based descriptions and formal relations, a combination of a textual representation and a graphical

overview should be created. We advocated for textual template-based descriptions regarding its handover facility for requirement engineers used to textual requirement definition. Coupled to formal relationships, their work should be facilitated to make decisions regarding alternatives with a clear understanding of the known impacts. A fully textual description of these impact relationships is not always intuitive, especially to identify them at a glance, so that the advantages brought by this formalization could be lost by the lack of conciseness and clarity of large textual models.

In the present chapter, and as we will detail in Chapter 6, the graphical overviews must be maintained separately from the textual models. As for **DAD** models, a two-way synchronization mechanism between textual and graphical representation should be investigated to help requirement engineers and architects to identify refinements, alternatives and exclusions/impacts more intuitively.

4.6 Formalizing the Requirements, Rationale and Decisions for the Online Library

We will now illustrate the **ASR** modeling constructs with the online library system introduced in the preceding chapter. We first translate the use case diagrams presented in Figure 3.8 in Section 3.7 into an `ASR` model. Afterwards, we will exemplify the available relationships between **Asr** and other **DAD** constructs.

4.6.1 High-Level Requirement Definition

We translated the use case diagrams of the online library system into the `ASR` model presented in Figure 4.4. We used the same identifying names (but in a *concatenated* version) for all requirements as the ones defined in the use case diagrams.

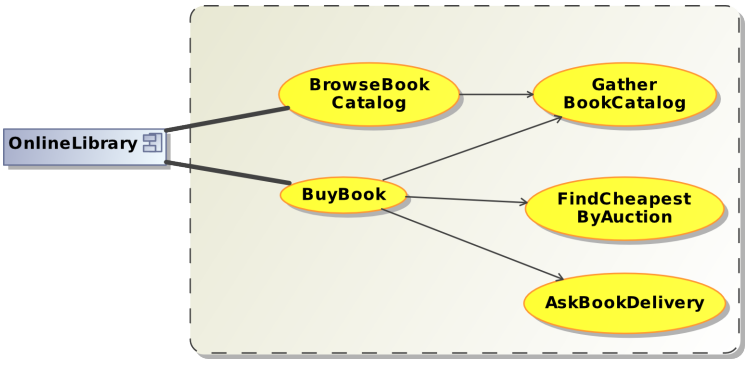


Figure 4.4: **Architecturally Significant Requirement** model for the `OnlineLibrary`

The **ASR** graphical representation depicted in Figure 4.4 contains five functional requirements that correspond to the five use cases presented in the previous chapter. At this level, the interactions between the components is not visible, like we did in

the use case diagram (even if this representation is not very common, but accepted by the UML standard). As we will discuss later in this section and in Chapter 7, these interactions may appear in **ASR** models, at certain point of development, with, for example `assignment` decisions or `Interface Usage` or `Implementation`. The complete model expressed in the textual syntax is reproduced in Listing 4.2.

```
package be.iodass.onlinelibrary;
asrmodel onlinelibrary_hl with be.iodass.onlinelibrary.onlinelibrary_naive {
  func BrowseBookCatalog assigned OnlineLibrary {
    description "The OnlineLibrary shall display a catalog of books to Customers.";
  }

  func BuyBook assigned OnlineLibrary {
    description "The OnlineLibrary shall sell books to Customers.";
  }

  func GatherBookCatalog assigned OnlineLibrary {
    description "The OnlineLibrary shall gather the books from all BookStores and
                 display them to Customers.";
    refines BrowseBookCatalog {
      assessment "To display the catalog, the library must first retrieve them.";
    }
    refines BuyBook {
      assessment "The library must compile all catalogs from the stores.";
    }
  }

  func FindCheapestByAuction assigned OnlineLibrary {
    description "When the Customer buys a book, the OnlineLibrary shall determine
                 the selling price by an auction with all BookStores.";
    refines BuyBook {
      assessment "The sell price is determined by an auction between the stores.";
    }
  }

  func AskBookDelivery assigned OnlineLibrary {
    description "When the cheapest price for a book is found, the OnlineLibrary shall
                 send the Customer and BookStore details to the ParcelDelivery.";
    refines BuyBook {
      assessment "Delegate the responsibility to the deliverer.";
    }
  }
}
```

Listing 4.2: **ASR** model in textual syntax

In the textual representation, the requirements are fully defined with a textual description that follows the **xEARS** templates, together with the refinement relations and more details about the split into lower level requirements. For now, we simply translated the main objectives of the library systems into a more structured model. In the next section, we will illustrate the other **ASR** modeling constructs in a larger model where we refine the *"GatherBookCatalog"* `Asr`.

4.6.2 Adding Alternatives and More Rationale in ASR Models

The **ASR** formalism offers the possibility to define a wide range of relations between requirements. In Figure 4.5, we give a larger overview of the available types of relations between requirements.

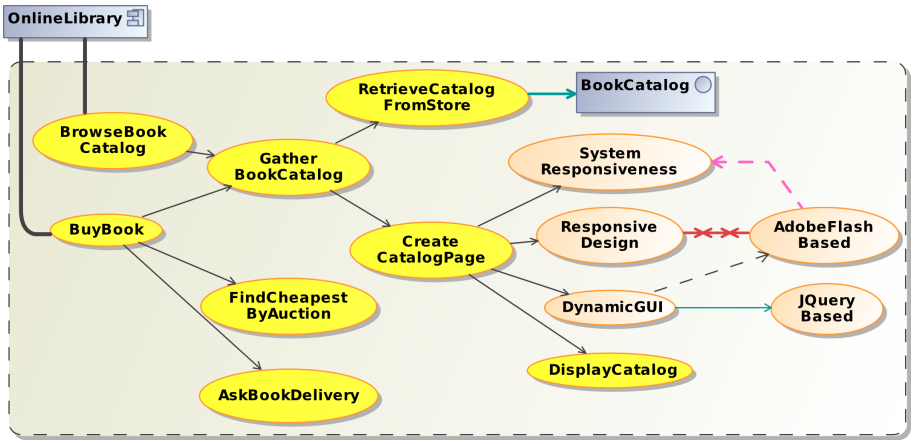


Figure 4.5: More detailed **ASR** model for the `OnlineLibrary`

In the above model, the requirement *"GatherBookCatalog"* has been refined into two lower level functional requirements: *"RetrieveCatalogFromStore"* and *"CreateCatalogPage"*. Assuming that the `BookCatalog` `Interface` already exists in the referenced **DAD** model, the *"RetrieveCatalogFromStore"* `Asr` will *use* it (depicted by the directed thick turquoise blue arrow). The other `Asr`, *"CreateCatalogPage"*, is refined in three non-functional requirements to specify the intended qualities of the webpage. The *"DynamicGUI"* is also refined into alternatives (depicted by dotted arrows) from which the *"JQueryBased"* is selected (directed thin turquoise blue arrow). The other alternative, *"AdobeFlashBased"* is declared in mutual `Exclusion` with the *"ResponsiveDesign"* (thick red line), since Adobe® Flash® is a graphical rendering technology that is currently not well supported by portable devices. Also, the same `Asr` has a negative `Impact` on the *"SystemResponsiveness"*, because it is a rather greedy technology (depicted by the pink dotted arrow).

In Listing 4.3, we detail the **ASR** model depicted in Figure 4.5.

```
1  package be.iodass.onlinelibrary;
2  asrmodel onlinelibrary with be.iodass.onlinelibrary.onlinelibrary {
3
4    func BrowseBookCatalog assigned OnlineLibrary {
5      description "The OnlineLibrary shall display a catalog of books to Customers.";
6    }
7
8    func BuyBook assigned OnlineLibrary {
9      description "The OnlineLibrary shall sell books to Customers.";
10   }
11
12   func GatherBookCatalog assigned OnlineLibrary {
13     description "The OnlineLibrary shall gather the books from all BookStores and
14                  display them to Customers.";
15     refines BrowseBookCatalog {
16       assessment "To display the catalog, the library must first retrieve them.";
17     }
18     refines BuyBook {
19       assessment "The library must compile all catalogs from the stores.";
20     }
```

```
21  }
22
23  func RetrieveCatalogFromStore assigned OnlineLibrary {
24    description "The OnlineLibrary shall retrieve the catalogs from all BookStores.";
25    refines GatherBookCatalog {
26      assessment "Concentrate on communication to bookstores.";
27    }
28    uses BookCatalog {
29      assessment "Reuse existing interface proposed by bookstores.";
30      strength "No needed extra development.";
31      weakness "Dependency to external changes, no hands on it.";
32    }
33  }
34
35   func CreateCatalogPage assigned OnlineLibrary {
36    description "The OnlineLibrary shall create a webpage with an aggregated view of
37                 all catalogs retrieved from the BookStores.";
38    refines GatherBookCatalog {
39      assessment "Build webpage from retrieve catalogs.";
40    }
41   }
42
43  nonfunc SystemResponsiveness assigned OnlineLibrary {
44    description "The OnlineLibrary shall be responsive to Customer requests.";
45    refines CreateCatalogPage {
46      assessment "Responsiveness is an important feature for today's webpages.";
47      constraint responsetime : 1000; "response time should stay beyond the second";
48    }
49  }
50
51  nonfunc ResponsiveDesign assigned OnlineLibrary {
52    description "The OnlineLibrary shall offer a responsive design for the book catalog
53                 to Customers.";
54    refines CreateCatalogPage {
55      assessment "Mobile applications/browsing must be taken into account too.";
56    }
57  }
58
59  nonfunc DynamicGUI assigned OnlineLibrary {
60    description "The OnlineLibrary shall offer a dynamic and eye-candy
61                 graphical interface to Customers.";
62    refines CreateCatalogPage {
63      assessment "Dynamic UI is a must for web pages.";
64    }
65  }
66
67  nonfunc AdobeFlashBased assigned OnlineLibrary {
68    description "The OnlineLibrary shall use Adobe Flash technology for the catalog
69                 webpage.";
70    refines CreateCatalogPage is alternative {
71      assessment "Adobe Flash is trendy";
72      weakness "Pretty greedy on CPU and not fully supported on all platforms";
73    }
74    exclude ResponsiveDesign {
75      assessment "Flash technology currently not supported in all (mobile) platforms.";
76    }
77    impact negatively SystemResponsiveness {
78      assessment "Client devices can suffer from responsiveness problems (updates or
79                  low-end CPUs).";
80    }
81  }
82
83  nonfunc JQueryBased assigned OnlineLibrary {
84    description "The OnlineLibrary shall use JQuery-based technology for the catalog
85                 webpage.";
86    refines CreateCatalogPage is selected {
87      assessment "JQuery is a reference technology based on javascript and ajax.";
```

```
88        strength "Library well tested and easy to use.";
89        weakness "Exclude 'disabled-javascript' webbrowsers.";
90      }
91    }
92    /* skip remaining of file with FindCheapestByAuction and AskBookDelivery */
93  }
```

Listing 4.3: More Detailed **ASR** model in textual syntax

This **ASR** model is bound to the `onlinelibrary` **DAD** model given in Listing 3.3 in Section 3.7.3. A set of new requirements are introduced in this model from the *"GatherBookCatalog"* `Asr`, starting at line 24. The *"RetrieveCatalogFromStore"* refines the *"GatherBookCatalog"* `Asr` and *uses* the `BookCatalog` `Interface`. For each of these `DesignDecisions`, at least an explanation is given in the `Assessment` clause and the `Usage` is further argued with a `Strength` and a `Weakness`. In the listing, we also illustrate the usage of a structured property for a `Constraint` at line 47 to precisely define the maximum response time of the `OnlineLibrary` regarding its responsiveness for the *"CreateCatalog"* `Asr`. The two last requirements, from line 67, illustrate the `Impact` and `Exclusion` mechanism. Again, such decisions must be enriched with some `Rationale`. Note that alternative `Asr`, *i.e.* possible choices regarding a design decision, are identifiable by the keyword `is alternative` and are pointing to their *higher-level* requirement. The selected alternative is annotated by `is selected`.

4.6.3 Concluding Remarks on the Examples

We do not illustrate all relations or types of rationale in the above samples, but the mechanisms are analogous to the ones exposed here. The textual syntax used for the definition of `Asr` has been designed as simple as possible, with few keywords, and remains understandable by non-practitioners with rather few effort. As **DAD** models, user-defined properties can be included to add meta-informations or, as illustrated in Listing 3.3, to formally specify some `Rationale`.

Also, **ASR** models may import other **DAD** or **ASR** models too. Externally defined patterns may be imported to fulfill a specific requirement or a large system can be split into multiple **ASR** files, each of them related to an identifiable part of the overall system, for example.

However, we do not expose the `Realization` relationships in the current samples. As defined in Section 4.3.2, `Realizations` are complex structural changes that must be applied to a **DAD** model in order to implement an `Asr`. These changes must be expressed as model transformations and will be the subject of Chapter 5.

4.7 Wrap-Up and Conclusions on ASR Modeling

In this section, we introduced the **Architecturally Significant Requirement** (**ASR**) modeling language. Combined to their textual specifications, system engineers are able to trace the relations between them as well as to record the design rationale behind each decision in a structured and straightforward manner. The

available relations and types of rationale have been defined on top of existing academic research [Kruchten et al., 2006; Zimmermann et al., 2009] and similar concepts are available in academic and industrial architectural knowledge management tools [Tang et al., 2006; Jansen et al., 2007; Burge and Brown, 2008; Bracewell et al., 2009].

However, the current modeling language has no explicit support for business goals, but the `Refinement` relationship can be used to that purpose to bind organizational goals to system requirements. This extension would need to be coupled to user-defined properties in order to describe other types of requirement than architecturally significant ones. For example, one may define its own set of properties to specify subtypes of requirements.

A second extension to the current language would be to add the ability to define an order between requirements in case of sequential executions of requirements. The simplest manner, but far from optimal, would be to reuse the *priority* attribute to that purpose. But a more generic strategy would be to ask requirement engineers to follow similar patterns as the extended EARS templates presented in Section 4.5.1 and extract from the textual description of `Asr` these kind of relations. A simple technique would consist in first extracting the *« is preceded by »* relations from preconditions and trigger expressed in the first part of the **xEARS** sentences, and in a second pass over the **ASR** model, add the opposite *« is followed by »* relations to the involved requirements.

To sum up, with **ASR** models, requirement engineers are able to record requirements with their relationships and design rationale together with an architectural model, since a particular **ASR** model is always bound to a **DAD** model. The combination of both models gives a snapshot of the identified architectural requirements and the corresponding architectural model at a given state of development.

CHAPTER 5

HOW TO INJECT NEW CONCERNS BY MODEL TRANSFORMATIONS

*In the present chapter, we now propose an ad-hoc concrete syntax-based transformation language to manipulate **DAD** architectural models. We first consider the applicability of using transformation languages as first-class entities in architecture design activities. We also discuss the relevance to define a custom transformation language directly on concrete model elements instead of reusing an abstract syntax-based transformation framework. Afterwards, we consider the assets of architectural styles and patterns and their roles in system architecture design and evolution. We then detail the available transformation rules to manipulate **DAD** models and, finally, illustrate the approach with our online library system.*

5.1 Why Do We Need a Transformation Language ?

The two preceding chapters introduced the core modeling languages of our architectural framework. The first language focuses on structural descriptions of software architectures around three inter-related layers. The second language is dedicated to record architecturally significant requirements with design decisions and rationale.

In this dissertation, we pointed many times the maintenance and evolution problems of software architectures that tend to become too obscure to accept changes,

as time goes by. In the present and following chapters, we will argue that a formalized architectural design and change management will help to keep control over an architectural representation on the long run.

Before going into detail over the proposed design framework in Chapter 6, we first introduce a custom transformation language working on the concrete syntax of **DAD** modeling elements. We rely on a transformation language to apply changes in a predefined, structured and reproducible manner. Coupled to an adequate documentation, likewise Jansen and Bosch's view on *« software architecture as a set of architectural design decisions »* [Jansen and Bosch, 2005], we use model transformations to formally record architectural changes as decisions implementations.

Since in the current dissertation, we raise model transformations as primary means to draw and modify architecture models, we decided to create an ad hoc transformation language working on the concrete syntax directly. As we discussed in Section 1.4, several model transformation techniques exist, articulated around four paradigms: *imperative*, *declarative*, *graph-based* and *concrete-syntax-based*. We highlighted the main assets of all types of languages, with at least one particular language of each paradigm and we argued for the most appropriate approach to use in our framework.

We came to the conclusion that we need a concrete syntax-based formalism for two main reasons. First, we intend to lower the learning curve for a new language with complex notions such as meta-modeling or higher-order transformation rules, so that practitioners stick to architectural concepts only. Second, styles and patterns play a key role in the software architecture field because they offer reusable solutions to common problems or provide effective *technical* answers to frequent needs for architectural qualities. Instead of describing a pattern only structurally, the **DAD-Transformation** language allows to specify them in some kind of *instantiation* rules so that they can be imported directly as model transformations.

As an introduction, we will depict a simple client-server architecture that will be partially created by model transformations. Then, we will concentrate on the roles played by styles, pattens and model transformations for common problems in the software engineering field, at first for reusability problems, then for requirement traceability and architecture maintenance.

5.2 An Introductory Sample: Build a Client-Server by Model Transformations

To give an overview of our concrete syntax-based transformation language, we will create a simple *Client-Server* architecture with model transformations. First, in Listing 5.1, we define two empty `ComponentTypes`, a `Client` and a `Server`, without any connection between them. They will be bound later by dedicated transformations rules.

```
1 package be.iodass.sample;
2
3 dadmodel clientserver {
4   definition {
```

```
    componenttype Client { }
    componenttype Server { }
  }
}
```

Listing 5.1: Sample *Client-Server* architecture model

Then, in Listing 5.2, we create their respective `Facets`, typed by a common `Hello Interface`, also created by a transformation in the same code snippet.

```
package be.iodass.sample;

/* some details are hidden for the sake of simplicity */

transformationset myfirsttransformation /* some details hidden here */ {

  // create interface
  create interface Hello {
    sync void hello();
  }

  // create facets on each part
  alter componenttype Client{
    uses Hello as hello;
  }
  alter componenttype Server {
    implements Hello as hello;
  }

  // create ConnectorType
  create connectortype One2One { mode one2one; }

  // create binding
  create linkagetype from Client.hello to Server.hello with One2One;
}
```

Listing 5.2: Sample transformation rules to created a typed connection in the *Client-Server* architecture

The resulting architecture model now contains the created Interface and both `Client` and `Server` are connected using that `Hello Interface`.

```
package be.iodass.sample;

dadmodel clientserver {
  definition {
    componenttype Client {
      uses Hello as hello ; // used facet correctly created
    }
    componenttype Server {
      implements Hello as hello ; // implemented facet created too
    }

    // new model elements created
    interface Hello { sync void hello (); }
    connectortype One2One { mode one2one; }

    // new linkage type created
    linkagetype from Client.hello to Server.hello with One2One ;
  }
}
```

Listing 5.3: Resulting *Client-Server* architecture model

In the present chapter, we will specify how modelers may influence architecture models through formal model transformations. But, first, we will discuss about the links between styles, patterns, model transformations and system architectures.

5.3 Styles, Patterns, Transformations and Architectures

The usage of patterns has been pushed forward for many years in the software engineering field. In the present section we first go back to the roots of design patterns and we list their core assets for software development. We then discuss the current problems in pattern reusability and how we intend to address this issue. We finally discuss the possible benefits of formalized model transformations in terms of traceability, maintenance and evolution of architecture models.

5.3.1 What Are Patterns and Styles, and What Are They Used For ?

According to many authors, the origin of the interest in design patterns traces back to a series of books mainly written by the professor and architect Christopher Alexander [Alexander, 1999]. In their *"A Pattern Language - Towns, Building, Construction"* [Alexander et al., 1977], the authors defined a list of 253 architectural patterns, when interconnected to each other, specify a whole language to construct towns and buildings. According to the authors, « *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.* » Each pattern is described following exactly the same structure with an illustrative picture, an introductory paragraph, the description of the problem and solution, a schematic diagram of the pattern and finally the relations to the other patterns.

In a seminal paper, Gamma *et al.*, *aka* the *"Gang of Four"*, defined patterns as a mean to « *capture intent behind a design by identifying objects, their collaborations, and the distribution of responsibilities* » [Gamma et al., 1993]. In a very similar fashion as *generics* in many programming languages, also defined as *parameterized types*, where object types are specified in a generic and reusable manner with other types as parameters, patterns are parameterized structures that may be reused in a wide range of domains. They usually offer structural or behavioral *principles* to recurrent problems. They abstract a whole architectural knowledge, with its rationale, decisions and consequences [Harrison et al., 2007]. The scientific literature abounds in pattern reference books over various domains like, among others, organizational processes [Coplien, 1994], software architecture [Buschmann et al., 1996], enterprise architecture [Fowler, 2002], information visualization [Heer and Agrawala, 2006] or service-oriented applications [Daigneau, 2011].

From the interest born from the original paper, Gamma *et al.* concentrated in an influential book named *"Design Patterns, Elements of Reusable Object-Oriented Software"*, a list of object-oriented patterns using a very detailed structure. In an average of ten pages, a pattern was specified by its *intent, motivation, applicability, structure, participants, collaborations, consequences, implementation, sample code,*

known uses, and related patterns [Gamma et al., 1995]. At the opposite to this very meticulous description, the *Portland* pattern paragraph[1] is a text-based form with a few sentences describing the problem and a few lines depicting the solution, separated by the term *"therefore"*. Many other forms exists in the literature, like in the aforementioned articles or books, such that, no real consensus exists in the way of specifying patterns.

In Section 1.3, we already stressed the relevance of design decisions in architecture design activities. Encompassing Jansen ad Bosch's view on *software architectures as a set of design decisions* [Jansen and Bosch, 2005], Taylor *et al.* articulate architectural patterns and styles around architectural design decisions [Taylor et al., 2009]. In their book, they defined a style as *« a named collection of architectural design decisions that (1) are applicable in a given development context, (2) constrain architectural design decisions that are specific to a particular system within that context, and (3) elicit beneficial qualities in each resulting system. »* Similarly, a pattern is *« a named collection of architectural design decisions that are applicable to a recurring design problem, parameterized to account for different software development contexts in which that problem appears. »* In the authors' view, styles and patterns are expressed in terms of decisions, which may comprise architectural configurations, *i.e.* models, but not only. Other guidelines and constraints may also be expressed in an informal manner to raise the level of genericity as much as possible.

Even if both definitions look very similar, they differ in their applicable scope and their abstraction level. Patterns are specified for a particular problem where styles apply to a particular context. Also, styles are often subject to human interpretation since they depict general guidelines instead of parameterized architectural blocks, as patterns do. In a very shortened way, styles constrain a whole architecture where patterns offers reusable parts to build that architecture.

According to Prechelt *et al.* and as commonly claimed in the software engineering practice [Prechelt et al., 2002], design patterns **(i)** improve productivity and software quality, **(ii)** help juniors to develop their design skills, **(iii)** promote best practices and **(iv)** support the communication between practitioners. In a rather wide literature review, Zhang and Budgen identified that *« the variety of form and scope that arises means that "blind" application of patterns with any sense of the potential limitations is unwise »* [Zhang and Budgen, 2012]. On top of methodological recommendations to define and evaluate the usage of patterns, they observed a lack of documentation over why a specific pattern was chosen to tackle a given problem, so that *juniors* had troubles to understand the system design. In other words, without an appropriate documentation, it gets hard to understand the purpose of a design pattern, then to re-apply it to other similar concerns. On the other side, if the documentation is too specific, the capability to reuse a pattern for a similar problem, but in a distinct domain is almost void. That is what we call the *genericity and reusability problem*.

[1] pattern description form used by three authors from Portland (Oregon, United States) at the *Pattern Languages of Programs conference* in 1994, see http://c2.com/ppr/about/portland.html

5.3.2 The Genericity and Reusability Problem

However, to make them reusable across people, projects and domains, some important pieces of information are needed, especially for novice designers with few knowledge over a domain or a problem. In *Schema-based* approaches inspired by the artificial intelligence community [Minsky, 1974; Rich, 1981], « *schemata can represent knowledge at all levels-from ideologies and cultural truths to knowledge about the meaning of a particular word, to knowledge about what patterns of excitation are associated with what letters of the alphabet. We have schemata to represent all levels of our experience, at all levels of abstraction. Finally, our schemata are our knowledge. All of our generic knowledge is embedded in schemata.* » [Rumelhart, 1980] According to the knowledge schema theory transposed into the software development field, practitioners may develop their knowledge around semantic structures that may be combined to build software from these generic concepts instead of concrete solutions [Détienne, 1990].

As defined in the above section, patterns are made to conceptualize concrete solutions into abstract and reusable concepts. As Zhang and Budgen, we believe patterns may potentially fulfill the role of knowledge schemata for the software architecture domain if they are specified with an appropriate formalism, such that patterns may be reused crosswise domains. By appropriate formalism, and closely to other pattern definition styles, we advocate for a combined structural definition of patterns with architectural models and a semi-formalized text-based description to refine the semantics of the pattern. In our framework, we already introduced a modeling language to represent system architectures in Chapter 3 that we will reuse in the present chapter. On the other side, we proposed a semi-formal language to record architecturally significant requirements with structured design rationale and decision traceability.

In the present dissertation, we argue that the combination of both views may help in documenting the pattern in a reusable manner with the following pieces of information:

(1) a structural description in a platform independent formalism, *i.e.* a **Definition-Assemblage-Deployment** model with a meaningful name
(2) a description of the problem it tackles, *i.e.* an **Architecturally Significant Requirement** model
(3) an argumentation of the rationale behind the pattern in the **ASR** model
(4) the links to other patterns, if any, through their related **ASR** models

Indeed, the genericity of the patterns defined that way highly depends on the ability of the modelers to specify them in a reusable manner. Transformations should be defined in a *self-contained* document, with abstract elements, exactly as in any other pattern formalism. When a pattern is inserted into a specific model, mapping rules, i.e. transformations, will bound the abstract model element to existing architectural elements. Examples of such pattern definitions will be illustrated in Sections 5.5.2 and 5.6.2.

The coupled architectural and requirement models provide a structured mechanism to formally define patterns, as well as architectural styles. Their usage condi-

tions may be fully described with details regarding, on the one hand, the benefits, weaknesses, constraints and hypotheses of using the pattern, and on the other hand, the impact to other patterns, like mutual exclusions, for example. Furthermore, because of the flexibility of both languages, patterns may be specified with very abstract components linked by semantic-less usage relationships associated to a very generic textual description. At the opposite, very detailed component structures with domain-specific concerns may also be expressed. The description style may depend on the modeler's wish or on project-specific requirements, for example.

5.3.3 Traceability, Maintenance and Evolution

In Section 1.3, we discussed about many research focusing on architectural knowledge documentation and design decision traceability. Both problems have been identified to be crucial, also at the eye of the industry. In the preceding chapter, we specified a requirement formalism to raise traceability links concerning a set of design decisions as first class entities in system architecture development. However, complex structural changes in such architectural models cannot be represented as simple decisions as the ones we detailed in Chapter 4. Applying patterns or architectural styles into models are part of the job, but custom modifications must be applied at some point of the development, maintenance or evolution of a software system.

In our proposal, we suggest to express any change as model transformations to reinforce once more the traceability links of the subsequent injections of new concerns into an architecture model. The traceability of these modifications are materialized by the `Realization` links in **ASR** models we defined at the end of Section 4.3.2. Exactly as any other type of `DesignDecision`, the `Realization` link may be further justified with any type of `Rationale`.

The second part of the traceability is gathered in a transformations set were architects formally define the structural changes they want to apply to a model. This way, architects, or any involved stakeholder with a basic understanding of the **DAD** formalism, is able to retrieve the structural changes made into the model in order to implement a new requirement. A full traceability of the successive changes is kept for later reference or, even *rollback* to older versions of the model, if needed. Also, alternative styles or patterns may be evaluated concurrently by modelers without losing previous versions of a model, or even under some conditions, reuse a set of transformations that has been defined in some other explored alternative solutions.

5.4 Manipulate DAD Models with Model Transformation

In order to balance the effort induced by our stringent mechanism to make changes in architectural models, a dedicated concrete-syntax based language has been defined. It reuses the same logic and, wherever possible, the same syntax as the **DAD** textual formalism presented in Section 3.7. We put some effort in making the syntax and transformation rules as concise and *self-explaining* as possible.

5.4.1 Overview of DAD Transformation Rules

Likewise the **DAD** and **ASR** languages, we summarize the main concepts of the **DAD-Transformation** (**DAD-T**) language, in the meta-model depicted in Figure 5.1[2].

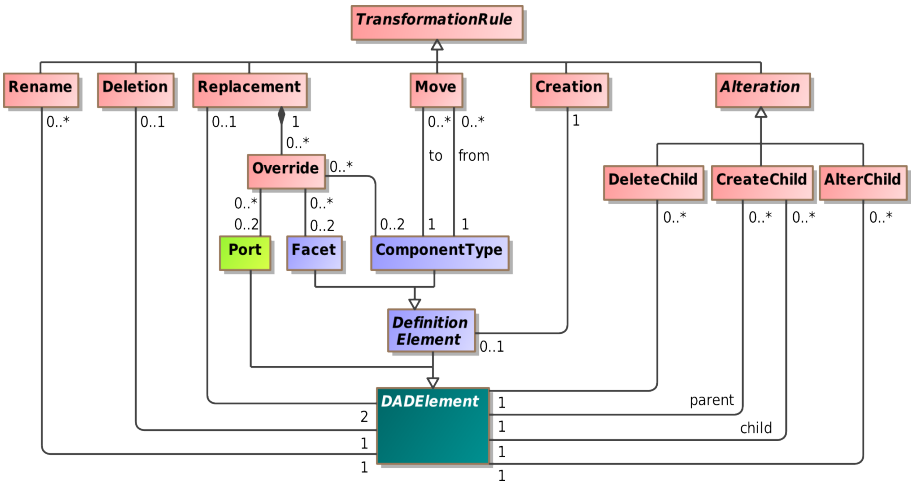


Figure 5.1: **DAD-Transformation** meta-model

Six types of `TransformationRules` may be defined on `DAD` models: renaming, creation and deletion of elements, fine-grained alternation of composite structures (`ComponentTypes`, `Interface`, `DataStructures` or `SetOfInstances`), moving of `ComponentTypes`' children, and replacement of model elements by other ones. We detail in the following sections all types of transformations as well as the verifications done on the produced models [3].

5.4.2 Creation Rules

Many types of constructs can be created with dedicated transformations rules. All **Definition** structural model elements, as presented in Section 3.2.2 and the three types of `GenericTypes`, can be added into a model with `creation` statements. Namely, these elements are `GenericTypes`, `ComponentTypes`, `Protocols`, `LinkTypes`, `NodeTypes`, `MediumTypes`, `GateTypes`, `LinkageTypes` and `LeakUsages`.

For all these types of elements, except `ComponentType`, they are created at the root of the target **DAD** model. `ComponentTypes` may be created as children of a parent `ComponentType`, since they can be composite.

[2] Note that for clarity reason, many `TransformationRules` are linked to a `DADElement`, but some restrictions exist for some transformations. These particular cases will be detailed in the following sections.

[3] The complete grammar is provided in Appendix D.

Inner configurations of `ComponentTypes`, i.e. `LinkageTypes` or `LeakUses` between children `ComponentTypes`, are directly created using their fully qualified names[4].

All `creation` transformations use the *create* keywords followed by the definition of the modeling elements expressed in **DAD** notation. For `ComponentTypes`, the rule may be followed by the reference to a parent where the newly created element will be placed. Listing 5.4 illustrates the `creation` of a `ComponentType` in an existing parent[5] with one `Facet` implementing a previously defined `Interface`.

```
1 create componenttype Child {
2   // example facet that reference an interface defined elsewhere
3   implements AnInterface as aFacet;
4 } parent Father;
```

Listing 5.4: Creation of a `ComponentType` inside a parent `ComponentType`

Newly created elements can be directly referenced in the same transformations set. For example, two `ComponentTypes` can be directly used to create a `LeakUses` dependency, as illustrated in Listing 5.5.

```
1 create componenttype ct1 { /* empty */ }
2 create componenttype ct2 { /* empty */ }
3 create usage from ct1 to ct2;
```

Listing 5.5: Creation of a two `ComponentTypes` and a dependency between them

5.4.3 Creation Verifications

For every newly created element, a check is performed on the uniqueness of the fully qualified name. When the transformation engine will execute the **DAD-T** set, it will apply the transformations as it reads them, so in the same order as they were defined by the modeler. If a new element is created with an existing fully qualified name, the transformation will be aborted[6].

For `LinkageTypes`, the same verifications as the ones exposed in Section 3.2.7 are also performed to ensure the compatibility between bound `Facets`. Again, if any error occurs, the engine will stop and feedback messages will be displayed to the modeler.

5.4.4 Renaming Rule

Model elements can be renamed by `renaming` rules. Any structural **Definition** layer construct, i.e. `GenericType`, `ComponentType`, `Protocol`, `LinkType`, `NodeType`, `MediumType` or `GateType`, may receive a new name. The target element must be referred by its qualified name, but only the new simple name must be specified in the second attribute of the transformation rule, as illustrated in Listing 5.6.

[4] in a *dotted* notation, as defined in Section 3.2.6

[5] the target **DAD** model is always implicitly imported.

[6] As we will discuss in Chapter 6, all these verifications will generate error or warning messages to users in the tool suite we provide next to this dissertation.

```
rename Father.Child as NewChildName;
```

Listing 5.6: Rename of a nested ComponentType

In this example, the Father.Child ComponentType will be renamed into Father. NewChildName. Obviously, every references to the old name will be updated accordingly, like for example in the LinkageTypes where it appears. Exactly as for creation rules, the transformation engine will avoid name clashes by aborting a transformations set that contains a conflicting rule. The same kind of user feedbacks will also be provided in such cases.

5.4.5 Deletion Rules

A large set of **DAD** model elements can be deleted. Among, this list, we first have the same constructs as the ones available for creation rules, *i.e.* GenericTypes, ComponentTypes, Protocols, LinkTypes, NodeTypes, MediumTypes, GateTypes, LinkageTypes and LeakUsages.

A deletion rule is simply denoted with the *delete* keyword and is represented in Listing 5.7 where the Child ComponentType is being deleted. For sub-components, the reference must always be expressed using their qualified name. The other elements, that cannot be nested, are referenced by their simple names. All deletion rule for these **Definition** elements are similar to Listing 5.7.

```
delete Father.Child;
```

Listing 5.7: Deletion of a nested ComponentType

Additionally, all **Assemblage** and **Deployment** layer elements, without any restriction, as well as Facets, may be deleted by a transformation. For those elements, the *delete* keyword must be followed by the type of the element to be deleted, written in lower case. An example is provided in Listing 5.8 where the Facet called aFacet of the Child ComponentType is deleted. Depending on the element to be deleted (nested or not), the deletion rule will need a simple or qualified name. In Listing 5.8, the Facet is referenced by its qualified name.

```
delete facet Father.Child.aFacet;
```

Listing 5.8: Deletion of a Facet

Last, for all types of relationships, which are unnamed model elements, the whole statement must be preceded by the *delete* keywords. We illustrate the deletion of a LinkageType in Listing 5.9. The other types of connections can be deleted in a similar fashion.

```
delete linkagetype from AComponentType.aFacet to AnotherOne.aFacet with ALinkType;
```

Listing 5.9: Deletion of a LinkageType

Finer-grained deletions are also possible on modeling elements defined in the *scope* of other *container* element, like Parameters that are nested into Services.

In these cases, we will not talk about `deletion` rules of the contained elements, but about `alterations` of their containers, as we will detail in Section 5.4.10.

5.4.6 Cascade Deletion Policies

All deletions are done *in cascade*, *i.e.* every modeling element *typed by*[7] the suppressed element will be deleted too. Additionally, every relationship where the deleted element is involved will be removed as well. The only two exceptions to this policy concern the `GenericTypes` that can involve some substitutions in `Service Parameters` and for the `Protocol` that does not trigger any other deletion.

The `deletion` transformation rule of a modeling element E is denoted by the procedure $delete(\mathsf{E})$. We reuse the same identifiers and conventions that the ones presented in Chapter 3.

Informally, when a `ComponentType` is deleted, all subcomponents are suppressed as well as all linkages or usages where this `ComponentType` appears. The deletion of the children `ComponentTypes` is done in priority in order to first delete recursively all contained `ComponentTypes`, their relationships to other elements and instantiations as `SetOfInstances`. Also, all `SetOfInstances` typed by this element are suppressed. We formally define the deletion for a `ComponentType` C as:

$$
\begin{array}{l}
delete(\mathsf{C}) \quad \{ \\
\quad \forall\, \mathsf{C}_i \mid \mathsf{C}_i \subset \mathsf{C} \quad : \quad delete(\mathsf{C}_i)\,; \\
\quad \forall\, \mathsf{F}_i \in \mathsf{C} \quad : \quad delete(\mathsf{F}_i)\,; \\
\quad \forall\, \mathsf{C}_i \mid \exists\, \mathsf{C} \curvearrowright \mathsf{C}_i \quad : \quad delete(\mathsf{C} \curvearrowright \mathsf{C}_i)\,; \\
\quad \forall\, \mathsf{C}_i \mid \exists\, \mathsf{C}_i \curvearrowright \mathsf{C} \quad : \quad delete(\mathsf{C}_i \curvearrowright \mathsf{C})\,; \\
\quad \forall\, \mathsf{S}_i \models \mathsf{C} \quad : \quad delete(\mathsf{S}_i)\,; \\
\}
\end{array}
$$

The deletion of a `Facet` includes the deletions of all related `LinkageTypes`. Additionally, any `Port` typed by this `Facet` is also removed, as we specify in the following:

$$
\begin{array}{l}
delete(\mathsf{F}) \quad \{ \\
\quad \forall\, \mathsf{F}_i, \mathsf{L}_k \mid \exists\, \mathsf{F} \xrightarrow{\mathsf{L}_k} \mathsf{F}_i \quad : \quad delete(\mathsf{F} \xrightarrow{\mathsf{L}_k} \mathsf{F}_i)\,; \\
\quad \forall\, \mathsf{F}_i, \mathsf{L}_k \mid \exists\, \mathsf{F}_i \xrightarrow{\mathsf{L}_k} \mathsf{F} \quad : \quad delete(\mathsf{F}_i \xrightarrow{\mathsf{L}_k} \mathsf{F})\,; \\
\quad \forall\, \mathsf{P}_i \mid \mathsf{P}_i \models \mathsf{F} \quad : \quad delete(\mathsf{P})\,; \\
\}
\end{array}
$$

[7] Remember the *typing* (namely *types* and *has_type*) relations depicted in Figure 3.1 from Chapter 3. For example, `SetOfInstances` are typed by `ComponentTypes`, or `Gates` are typed by `GateTypes`

Similarly to ComponentTypes, when a SetOfInstances is suppressed, all Linkages and LeakUsages where it appears must be deleted. Also, the Deploy statements of this SetOfInstances and all related Ports' Openings must be removed too. The $\overset{\curvearrowright}{\square}$ operator is overloaded to express usage dependencies between SetOfInstances too. The deletion of a SetOfInstance S is formally defined as follow:

$$
\begin{array}{l}
delete(\mathsf{S}) \quad \{ \\
\quad \forall\, \mathsf{P}_j \in \mathsf{S} \quad : \quad delete(\mathsf{P})\,; \\
\quad \forall\, \mathsf{S}_i \mid \exists\, \mathsf{S} \curvearrowright \mathsf{S}_i \quad : \quad delete(\mathsf{S} \curvearrowright \mathsf{S}_i)\,; \\
\quad \forall\, \mathsf{S}_i \mid \exists\, \mathsf{S}_i \curvearrowright \mathsf{S} \quad : \quad delete(\mathsf{S}_i \curvearrowright \mathsf{S})\,; \\
\quad \forall\, \mathsf{H}_i \mid \exists\, \mathsf{S} \hookrightarrow \mathsf{H}_1 \quad : \quad delete(\mathsf{S} \hookrightarrow \mathsf{H}_1)\,; \\
\}
\end{array}
$$

The deletion of a Port suppresses all involved Linkages as well as all related Openings:

$$
\begin{array}{l}
delete(\mathsf{P}) \quad \{ \\
\quad \exists\, \mathsf{L}, \exists\, \mathsf{P}_i \in \mathsf{S}_i \mid \mathsf{P} \overset{\mathsf{L}}{\frown} \mathsf{P}_i \quad : \quad delete(\mathsf{P} \overset{\mathsf{L}}{\frown} \mathsf{P}_i)\,; \\
\quad \exists\, \mathsf{L}, \exists\, \mathsf{P}_i \in \mathsf{S}_i \mid \mathsf{P}_i \overset{\mathsf{L}}{\frown} \mathsf{P} \quad : \quad delete(\mathsf{P}_i \overset{\mathsf{L}}{\frown} \mathsf{P})\,; \\
\quad \forall\, \mathsf{A}_i \mid \exists\, \mathsf{P} \circledcirc \mathsf{A}_i \quad : \quad delete(\mathsf{P} \circledcirc \mathsf{A}_i)\,; \\
\}
\end{array}
$$

Concerning platform-related types **Definition** and Deployment elements, the suppression of a NodeType N triggers the deletions of all typed Nodes, as we formally express in the following rule:

$$
\begin{array}{l}
delete(\mathsf{N}) \quad \{ \\
\quad \forall\, \mathsf{H}_i \mid \mathsf{H}_i \models \mathsf{N} \quad : \quad delete(\mathsf{H}_i)\,; \\
\}
\end{array}
$$

The deletion of a Node N triggers the suppression of all concerned Deploy statements, Ports Openings on the Nodes' Gates and Plugs in those Gates:

$$
\begin{array}{l}
delete(\mathsf{H}) \quad \{ \\
\quad \forall\, \mathsf{S}_i \mid \exists\, \mathsf{S}_i \hookrightarrow \mathsf{H} \quad : \quad delete(\mathsf{S}_i \hookrightarrow \mathsf{H})\,; \\
\quad \forall\, \mathsf{A}_i \in \mathsf{H} \quad : \quad delete(\mathsf{A}_i)\,; \\
\}
\end{array}
$$

The deletion of `GateType` G simply triggers the removals of all its `Gates` instances:

$$
\begin{array}{l}
delete(\mathsf{G}) \quad \{ \\
\quad \forall\, \mathsf{A}_i \models \mathsf{G} \quad : \quad delete(\mathsf{A}_i)\,; \\
\}
\end{array}
$$

The deletion of a `Gate` A suppresses all concerned `Openings` and `Plugs`:

$$
\begin{array}{l}
delete(\mathsf{A}) \quad \{ \\
\quad \forall\, \mathsf{P}_i \mid \exists\, \mathsf{P}_i \circledcirc \mathsf{A} \quad : \quad delete(\mathsf{P}_i \circledcirc \mathsf{A}_k)\,; \\
\quad \forall\, \mathsf{A}_k \mid \exists\, \mathsf{A} \overset{\mathsf{M}}{\smile} \mathsf{A}_k \quad : \quad delete(\mathsf{A} \overset{\mathsf{M}}{\smile} \mathsf{A}_k)\,; \\
\quad \forall\, \mathsf{A}_k \mid \exists\, \mathsf{A}_k \overset{\mathsf{M}}{\smile} \mathsf{A} \quad : \quad delete(\mathsf{A}_k \overset{\mathsf{M}}{\smile} \mathsf{A})\,; \\
\}
\end{array}
$$

Regarding architectural configuration **Definition** and **Assemblage** connections, for `LinkTypes` and `MediumTypes`, the same logic applies, all concrete connections using those elements are also suppressed. Formally, for `LinkTypes`:

$$
\begin{array}{l}
delete(\mathsf{L}) \quad \{ \\
\quad \forall\, \mathsf{F}_i, \mathsf{F}_j \mid \exists\, \mathsf{F}_i \overset{\mathsf{L}}{\rightharpoonup} \mathsf{F}_j \quad : \quad delete(\mathsf{F}_i \overset{\mathsf{L}}{\rightharpoonup} \mathsf{F}_j)\,; \\
\quad \forall\, \mathsf{P}_k \models \mathsf{F}_i,\ \forall\, \mathsf{P}_l \models \mathsf{F}_j \mid \exists\, \mathsf{P}_k \overset{\mathsf{L}}{\frown} \mathsf{P}_l \quad : \quad delete(\mathsf{P}_k \overset{\mathsf{L}}{\frown} \mathsf{P}_l)\,; \\
\}
\end{array}
$$

And for `MediumTypes`:

$$
\begin{array}{l}
delete(\mathsf{M}) \quad \{ \\
\quad \forall\, \mathsf{A}_i, \mathsf{A}_j \mid \exists\, \mathsf{A}_i \overset{\mathsf{M}}{\smile} \mathsf{A}_j \quad : \quad delete(\mathsf{A}_i \overset{\mathsf{M}}{\smile} \mathsf{A}_j)\,; \\
\}
\end{array}
$$

As we noticed in the beginning of the current section, two exceptions exist to this cascade policy: `GenericTypes` and `Protocols`. In case of `GenericTypes` deletions, any `Service` `Parameter` typed by this `GenericType` will be removed from the `Service` signature or set to a *void* type if it was a return `Parameter` value. Formally, if the `GenericType` g is deleted, then:

$$
\begin{aligned}
&delete(\mathsf{g}) \quad \{ \\
&\quad \forall\, s_i =< p_i^j > \wedge\, p_i^j = (d_i^j, g) \wedge\, d_i^j \neq return : delete(p_i^j)\,; \\
&\quad \forall\, s_i =< p_i^j > \mid \exists\, p_i^j = (d_i^j, g) \wedge\, d_i^j = return : p_i^j = (d_i^j, void)\,; \\
&\}
\end{aligned}
$$

For `Interfaces`, additional deletions must be triggered too for all implementing `Facets`. Formally,

$$
\begin{aligned}
&delete(\mathsf{I}) \quad \{ \\
&\quad \forall\, s_i =< p_i^j > \mid \left(\exists\, p_i^j = (d_i^j, g) \wedge\, g = \mathsf{I} \right) : delete(g)\,; \\
&\quad \forall\, \mathsf{F}_k \mid \mathsf{F}_k \models \mathsf{I} : delete(\mathsf{F}_k)\,; \\
&\}
\end{aligned}
$$

Finally, the second and last exception concerns the `Protocol`. Its deletion triggers no other subsequent deletions, so that any reference to this `Protocol` is simply lost. If a deleted `Protocol` was actually used in a `Linkage`, the modeler will be required to either replace the Protocol (see Section 5.4.7) or to modify the `Linkages` *a posterio*.

This cascade policy has been defined to simplify the work of modelers. Except for `Protocols`, they do not have to care about the links and references to the deleted element. All subsequent deletion rules will be calculated by the transformation engine.

5.4.7 Replacement Rules

Substitutions of many types of model elements can be performed on **DAD** models, possibly with overriding rules on specific contained elements, when necessary. `Replacement` transformations are particularly useful to replace an *API* or a *Component Off The Shelf* by another one or in case of highly decentralized systems that combine different parts developed by distinct teams. Old versions or compatible solutions may be replaced by newer ones with a few transformation rules. `Replacements` may only apply on `GenericTypes`, `ComponentTypes`, `Protocols`, `LinkTypes`, `NodeTypes`, `MediumTypes` and `GateTypes`.

A `replacement` rule always substitutes model elements of the same type, but their semantics can slightly vary. Informally, the *substitute* must at least provide the same *"features"* than the *target* element it replaces. This transformation is specified by the *replace* keywords, possibly refined with specific `override` rules (for containers) to explicitly define the matches between old contained elements to their substitutes. A simple replacement transformation is illustrated in Listing 5.10.

```
1 replace OldInterface by NewInterface keepname;
```

Listing 5.10: `Replacement` of an `Interface`

As *deletion* rules of **Definition** layer objects, old and new elements from a `replacement` rule are directly referenced by their (qualified) names, again to keep the syntax as light as possible. In the above rule, the `OldInterface` is replaced by the definition of the `NewInterface`. The *keepname* attribute states that the old name is conserved for the newly introduced model element. It is a kind of shortcut to the combination of a `replace` transformation immediately followed by a `rename` rule to the old name.

For `ComponentTypes` and `Interfaces`, `override` rules may be added to redirect children `ComponentTypes`, `Facets` and *instantiated* `Ports` to other compatible elements either defined or used inside the substitute. Overriding rules simply map elements defined in the target or referred from this target to other elements of the same type or with compatible definitions. Listing 5.11 illustrates a more complex `replacement` transformation with the definition of both the target and substitute `ComponentTypes`.

```
1  create interface AnInterface {
2    sync void do(in string something);
3  }
4  create interface AnotherInterface {
5    sync void doOther(in string other);
6    event someEvent();
7  }
8  create componenttype Target {
9    implements AnInterface as iface;
10 }
11 create componenttype Substitute {
12   implements AnotherInterface as otherIface;
13 }
14 replace Target by Substitute overrides {
15   facet Target.iface by Substitute.otherIface;
16 };
```

Listing 5.11: Complex `remplacement` rule

If both `ComponentTypes` contain exactly the same types of `Facets`, *i.e.* referring the same `Interface`, they will be implicitly matched to each other. But, if no obvious matches can be found, explicit overriding rules must be specified. In the above sample, both `Facets` are not identical, but they are compatible in our typing system since `AnotherInterface` contains a synchronized `Operation` that is compatible with the only `Operation` defined in `AnInterface`[8]. Analogously to the verification of the semantic fulfillment of a provided `Facet` that must at least provide the required `Services`, a replacement `Facet` must at least cover the `Services` to be substituted. Also, if the target `ComponentType` has been instantiated, its `SetOfInstances` will be now typed by the substitute `ComponentType` and their `Ports` will be updated accordingly to point to the newly `Facets`, if needed.

A slightly different overriding mechanism for `replacement` transformations may be defined for `Interfaces`. When an `Interface` is replaced by another one, every reference to this `Interface` is updated to point to the newly introduced substitute, except if specific overriding rules on `Facets` have been defined. Here with overriding rules, only the specified elements will be updated following these rules, otherwise

[8] remember we use a kind of *duck-typing* for `Interface` compatibility checks (see Section 3.2.7)

all typed Facets will be updated. Again, the existing LinkageTypes will be checked for composition compatibility after the transformation. Also, if an overriding rule creates a duplicate LinkageType, it will be implicitly removed.

To sum up, replacement transformations of all types of modeling elements, excluding Interfaces, are always performed for all references to this modeling element **and** for the specific overriding rules. At the opposite, replacement of Interfaces can be **constrained** by overriding such that the replacements will be performed only for the specified overrides. This method offers more flexibility for ComponentTypes compositions and Interfaces substitutions.

Finally, when substituting two ComponentTypes, both definitions can be merged into the replacement element, such that the old *inner-configurations* and Facets are kept inside the replacement ComponentType. This *merging* possibility is particularly interesting for pattern injections, as we will illustrate in Section 5.6.2.

5.4.8 Validity Checks on Replacements

Some verifications must be performed when replacing modeling elements by other ones. First, for Interfaces, and by extension Facets and ComponentTypes, similar verifications as the one performed for LinkageTypes presented in Section 3.2.7 are executed. The transformation engine will ensure that the substitute Interface matches the definition of its target, as well as the Port overriding rules, if any.

When replacing ComponentTypes and NodeTypes, every entry point *in-use* must also be checked. If a Facet or a Gate is actually used, *i.e.* part of a LinkageType or Plug, the substitutes must offer compatible entry points, otherwise the transformation is aborted. This verification is meant to avoid breaking a (platform) architecture model by introducing elements that will not provide the necessary connection facilities, as previously defined by their targets.

For LinkTypes and MediumTypes, the substitutes must still allow the existing connections to work properly. If there exist Linkages, LinkageTypes or Plugs on specific Protocols, the newly introduced elements must at least support these Protocols, otherwise warning messages will be issued. The same verification stands for Protocol replacements. For example, when a specific Protocol is used for a Linkage between two Ports, the substitute LinkType must obviously support this new Protocol or the LinkType must be updated to support it.

Finally, for any model element, in case of it has been refined with user-defined properties, additional checks may be performed. For ordered properties, the substitute model element should be tagged with at least the same or higher value for these properties. For unordered properties, the same value should be specified. If such mismatch occurs, warning messages are issued.

5.4.9 Moving Rules

At some point in the software architecture development or maintenance, a component must be moved inside a container to restrain its visibility or, at the opposite, it must be exposed as an individual component with its own life. In **DAD** models,

`ComponentTypes` may be moved inside other `ComponentTypes` or outside their parents, *i.e.* in the *root* of the **Definition** layer.

The `moving` rule can only be applied to `ComponentTypes` and has the following form, as depicted in Listing 5.12. To move a `ComponentType` inside another, the name of the new parent is simply specified instead of the *root* keyword.

```
1 move ChildComponent to root;
```

Listing 5.12: Move of a child `ComponentType` to the *root* of the model

Additional processing must be done when moving components from a container (the *root* is also a container) to another: all `LinkageTypes` must be updated accordingly, as well as all related `Ports` and `Linkages`. Informally, all `Facets` of the `ComponentType` must be checked to update the `LinkageTypes` that should be transformed from a `DelegationType` to a `ConnectorType` when the `ComponentType` is moved to the *root*, or to create new delegations (with the needed `Facets`) if it is moved inside another `ComponentType`. In the following formal definition, the `moving` rule is denoted by the $move(C_t, C_d)$ procedure to move a *target* C_t into a *destination* C_d. We only present the rules for `LinkageTypes` going to the target `ComponentType`, *i.e.* to its implemented `Facets`. Obviously, the other `LinkageTypes` are also updated following an identical process, but we hide them for conciseness reasons. Also, for readability reason, we split the different possible cases into four sub-rules. But the `move` transformation is a recursive process that combines all four cases.

$$
\begin{aligned}
&move(C_t, C_d) \quad \wedge \quad \exists\, C_p \mid C_t \subset C_p \wedge C_d \neq C_p \quad \{ \\
&\quad move(C_t, p(C_p)) ; \\
&\quad \forall\, F_t^i \in C_t,\, F_p^i \in C_p,\, L \mid F_p^i \xrightarrow{L} F_t^i \,:\, replace(F_t^i, F_p^i) ; \\
&\quad \forall\, F_i^i \in C_i \mid F_i^i \xrightarrow{L} F_t^i \,:\, C_p = C_p \cup \{proxy(F_t^i)\} ; \\
&\}
\end{aligned}
$$

First, the *move* procedure is called with the parent of the new container to raise recursively the target to either its destination or the *root*. Second, still recursively, the transformation will update the delegations for nested `ComponentTypes` that are raised up to their grand-parent container by bypassing the direct parent of the target. It will reconnect the `LinkageTypes` from the grand-parent directly to the grand-son by substituting the target `Facets` to the parent ones. The inner-configurations that involved the target will be updated too with *proxies* for the concerned `Facets` that will be added into the old parent `ComponentType`. These *proxies* are clones of `Facets` from the target `ComponentType` that will be bound to the inner-configuration `Facets`.

$$move(C_t, C_d) \quad \wedge \quad \exists\, C_p \mid C_d \subset C_p \quad \{$$
$$\quad move(C_t, C_p)\,;$$
$$\quad \forall\, F_t^i \in C_t, F_i^i \in C_i \mid \exists\, F_t^i \overset{\perp}{\rightarrow} F_i^i \,:\, C_p = C_p \cup \{proxy(F_t^i)\}\,;$$
$$\}$$

Second, when the `ComponentType` must be lowered down into a new container, new *proxies* must be created and added into the new container `ComponentType`. The content of the target must be moved to this new container too by calling recursively the *move* procedure and the required proxies will be created too.

$$move(C_t, C_d) \quad \wedge \quad C_d = root \quad \{$$
$$\quad p(C_t) = root\,;$$
$$\}$$

Third, if the destination is the *root*, the first part of the `moving` transformation where the `ComponentType` has been raised up from its original container have been already performed, so no other action must be done than setting the parent of the target `ComponentType` to the *root*.

$$move(C_t, C_d) \quad \wedge \quad p(C_t) = C_d \quad \{$$
$$\quad \forall\, F_t^i \in C_t \,:\, C_d = C_d \cup \{proxy(F_t^i)\}\,;$$
$$\}$$

Last, if the target is actually contained in the destination `ComponentType`, *i.e.* it has been recursively moved until either the child of the destination (with the first sub-rules) and/or to the direct parent of the destination (with the second sub-rules). Some needed proxies must still be created to let the target being accessible from inside the destination.

The combination of the two first sub-rules will converge to either the third or the fourth case such that the target will be moved from its original container to the destination and `Facet` *proxies* will be created at each *move* call.

5.4.10 Alteration Rules

In order to modify the content of some model elements, fine-grained transformation rules have been defined to alter the internal structure of these elements. To illustrate `alteration` rules, we will refer to the code samples given in Section 3.7 and modify some model elements introduced in the online library case study.

First, new `Facets` may be added into `ComponentTypes`[9], as illustrated in Listing 5.13 where the `OnlineLibrary` `ComponentType` created in Listing 3.2 now uses

[9] We already discussed the `Facets`' deletions in Section 5.4.5.

the AnotherInterface created in Section 5.4.7. The syntax is very close to the definition of a ComponentType only preceded by the keyword *alter*.

```
alter componenttype OnlineLibrary { uses AnotherInterface as another; }
```

Listing 5.13: Alter ComponentType to add new Facets

After the execution of the alter transformation, the Child ComponentType will now contain the new Facet, so that new LinkageTypes can be created for this Facet or corresponding Ports may be instantiated. These alteration rules offer a straight and traceable way to modify the semantics of structural model elements.

Gates may be added in a similar fashion to NodeTypes, again using the syntax as defined in **DAD** models.

Protocols may be added or deleted from the supported list in LinkTypes, MediumTypes or GateTypes. For any of these model element, the alteration transformation follows the same syntax, as illustrated in Listing 5.14[10] for the One2One ConnectorType and the GenericGatetype we defined in Listing 3.3.

```
alter connectortype One2One add HTTP;
alter gatetype GenericGatetype remove HTTP;
```

Listing 5.14: Alter list of Protocols in communication-related model elements

SetOfInstances can be fully altered, except for its related ComponentType and for Ports' deletion[11]. Changing the type of a SetOfInstances is only possible by replacement of its typing ComponentType. All other attributes may be changed with alteration rules, as shown in Listing 5.15 where we modify the library as specified in Listing 3.4.

```
alter soi library {
  rename biglibrary; // rename it
  card [1 10]; // change cardinality
  OnlineLibrary.another as another on HTTP; // create new port
  creates stores; // instantiates the stores
  destroys stores; // and destroys them
}
```

Listing 5.15: Alter the library SetOfInstances

With alteration rules, Nodes may be added or removed from a Site. It can also be renamed or its *situation* description may be updated. Listing 5.16 illustrates all possible modification of TheOffice Site introduced in Listing 3.5[12].

```
alter site TheOffice {
  rename TheExtendedOffice;
  situation "Its new extended situation";
  add newGateway;
  remove gateway;
}
```

Listing 5.16: Alter TheOffice Site

[10] Many Protocols can be added/removed in one rule, each of them separated by a *","* (*comma*).
[11] Already addressed in Section 5.4.5 with the deletion policy.
[12] As for alteration for accepted Protocols, many Nodes may be added/removed.

DataStructures can also be altered by adding, removing or replacing DataFields. These transformations offer a flexible way to update an existing structure, while keeping the history of the modifications made. The following Listing 5.17 illustrates the three types of modification on the CustomerDetails structure introduced in Listing 3.3.

```
alter struct CustomerDetails {
  add string nickname; // add a new field in structure
  replace number by string number; // change type of number
  remove country; // suppress 'country' field
}
```

Listing 5.17: Creation and modification of a DataStructure

Last Interfaces can be modified to add, remove or even rewrite partially or completely its Services. This alteration transformation has been defined for two main objectives: traceability and version control. First, an Interface is an interaction point through which ComponentTypes can communicate. If its semantics is modified, it is often valuable to know exactly what has been changed to update accordingly the behavior of the implementation or the other dependent software. This traceability can be fulfilled by such a transformation rule. Second, Interfaces' updates are sometimes related to new versions of a product. New releases of a software component can introduce new services or updated exceptions handling. Thus, a **DAD-T** set can play the role of an architectural patch between two versions of a (group of) Interfaces, for example.

In Listing 5.18, we alter the BookSelling Interface from Listing 3.3 to add exception handling to a Service and update the signature of another one.

```
alter interface BookSelling {
  add exception NoSuchBook(); // create a new exception
  rewrite buyBook {
    replace isbn by string isbn; // change type of isbn
    add float price; // add new parameter
  } // fully rewrite browseCatalog (name may also be changed, if necessary)
  rewrite browseCatalog by sync Book[] browseCatalog(in string genre) raises NoSuchBook;
}
```

Listing 5.18: Alteration of an Interface

As we detailed in the present section, both alterations for DataStructures and Interfaces are particularly useful for evolution purposes when some Services must be extended or new DataFields must be introduced or modified in an architecture model.

5.4.11 Alteration Verifications

When adding new Ports, a verification is performed to ensure the Port actually exists in the ComponentType, otherwise an error message will be displayed and the transformation will be aborted. In this scenario, a typing constraint is violated, so the transformation may not be executed.

When removing a `Protocol` that was used in an existing binding between model elements, a warning will be issued to the user. We use here the same policy as the exception defined for `Protocol` deletions in Section 5.4.5.

5.4.12 Summary of Available Transformations per Modeling Element

We recap now the model transformations we presented in the preceding sections. In Table 5.1, we summarize the existing transformations for all `DAD` elements. The first column lists the main DAD model elements, with first all **Definition** layer constructs, then the **Assemblage** elements and finally the **Deployment** ones. The other six columns gather the possible *effects* that can be expressed as dedicated transformations or as sub-rules from an `alteration` transformation. The table shows then the matches either between the **DAD-T** rules and the **DAD** elements, as well as the possible transformations from **alterations** of the container elements[13]. For example, a `Service` may not be created by a `creation` rule, but via an `alteration` of its containing `Interface`.

DAD Element	Create	Rename	Delete	Replace	Move	Alter
PrimitiveType	✓	✗	✓	✓	✗	✗
DataStructure	✓	✓	✓	✓	✗	✓
Interface	✓	✓	✓	✓	✗	✓
Service	✓	✓	✓	✓	✗	✓
Parameter	✓	✓	✓	✓	✗	✓
ComponentType	✓	✓	✓	✓	✓	✓
Facet	✓	✗	✓	✗	✓	✗
Protocol	✓	✓	✓	✓	✗	✓
LinkType	✓	✓	✓	✓	✗	✓
LinkageType	✓	✗	✓	✗	✗	✓
NodeType	✓	✓	✓	✓	✗	✓
GateType	✓	✓	✓	✓	✗	✓
MediumType	✓	✓	✓	✓	✗	✓
SetOfInstances	✗	✓	✓	✗	✓	✓
Port	✓	✗	✓	✗	✗	✗
Linkage	✗	✗	✓	✗	✗	✓
Site	✗	✓	✓	✗	✗	✓
Node	✗	✗	✓	✗	✗	✗
Open	✗	✗	✓	✗	✗	✗
Plug	✗	✗	✓	✗	✗	✗

Table 5.1: Summary of available transformations

As shown in the above table, any **DAD** element can be deleted from a transformation. All `Definition` layer elements can be created, some of them through

[13] Note we use verbal forms as column names to disambiguate from the **DAD-Transformation** rules.

`alterations` of their container (like `Services` or `Ports`). Almost all modeling elements may also be altered to update their definitions or their content according to architectural maintenance or evolution.

We decided to disallow the possibility to add or remove supertypes to `ComponentTypes` directly through `alteration` transformations. Instead, modelers will have to substitute a new `ComponentType` to the old one. We decided to use the `replacement` rule because, to us, such a modification resembles more to a substitution of architectural artifacts with different semantics than just an update of an existing semantics. Since transformation are meant to support an effective traceability of architectural changes, we selected that solution, even if it looks more verbose and stringent.

At current time, no alteration of properties assigned to model elements is possible. Even if such a mechanism could be integrated into the transformation language, this feature has been left out for one main reason. Properties are mainly useful for **Definition** layer elements that are specified once and may be reused many time. Adding a property with a dedicated transformation would be rather verbose and ineffective, in comparison of editing directly the definition of an element.

The only elements that cannot be created directly by a dedicated model transformation are either `Assemblage` or `Deployment` layer objects. We defined DAD-T sets as hybrid models to be able to specify Assemblage or Deployment clauses directly, as we will discuss in the following section.

5.5 DAD-T Set as a Hybrid Model

When creating new **Definition** layer elements, a set of **Assemblage** and/or **Deployment** statements must be specified. Besides, transformations are meant to be reusable to some extent. For these two reasons, the **DAD-T** language as been designed as a hybrid modeling facility with the possibility to add both latter clauses and to include other transformations defined in external **DAD-T** sets.

5.5.1 Assemblage and Deployment Clauses

Exactly as in a **DAD** model, any **Assemblage** and **Deployment** statement can be specified in a **DAD-T** set. The transformation language reuses both clauses natively and any statement will be imported in the target **DAD** model when the transformations defined in the **DAD-T** set will be applied by the engine. Listing 5.19 shows the general template of a **DAD-T** set.

```
1  package be.iodass.example;
2
3  asrmodel be.iodass.example.myasrmodel; // the involved asr model (mandatory)
4  dadmodel be.iodass.example.mydadmodel; // the involved dad model (optional)
5  // the transformation model name pointing to the concerned requirement
6  transformationset template concerns SomeRequirement {
7    /* there should be some transformation rules in here */
8    assemblage { /*  some new assemblage can be defined */ }
9    deployment { /*  some deployment mapping can be defined */ }
10 }
```

Listing 5.19: Template of a **DAD-T** set

A **DAD-T** model is always bound to one or more requirements, the `SomeRequirement` `Asr` in the above listing, specified in the model imported with the keyword *asrmodel* at line 3. Optionally, a **DAD** model may be imported too, when the transformations must be applied to elements defined in another model. This is actually the usual goal of a transformations set. However, as we will detail in the next section, a **DAD-T** set may be created as a standalone artifact, only bound to an **ASR** model that describes its objective.

Right after the list of transformation rules, **Assemblage** and/or **Deployment** statements can freely be defined in the model, so that the elements that cannot be created by dedicated transformation rules, can be specified in these clauses, exactly as in a **DAD** model.

Coupled to the close syntax of element `creations` or `alterations` we presented in the preceding sections, the amount of **DAD-T**–specific keywords remains low to minimize the needed additional work to learn those concepts.

5.5.2 Inclusion Mechanism

To enhance transformations reusability, **DAD-T** sets can be included in other sets in a simple manner. A dedicated *include* keyword is used to execute all transformations declared in the imported model. If **Assemblage** or **Deployment** clauses are also present in the included model, they will be added in the target model too, right after the **Assemblage** and/or **Deployment** statements of the main transformations set.

This inclusion mechanism is particularly useful for design pattern injection in **DAD** models. As an example, we will illustrate this method with the `Observer` pattern graphically depicted in Figure 5.2.

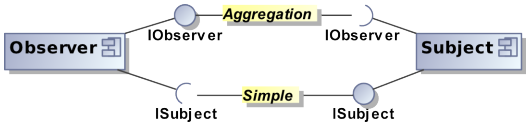


Figure 5.2: Representation of the `Observer` pattern in **DAD** graphical syntax

The above model corresponds to the following `DAD-T` set, reproduced in Listing 5.20.

```
1  package iodass.base.patterns;
2
3  asrmodel iodass.base.patterns.observer;
4
5  transformationset observer concerns Observer {
6    create interface IObserver {
7      async notify();
8    }
9
10   create interface ISubject {
11     async register(IObserver o);
12     async unregister(IObserver o);
13     async notifyObservers();
14   }
15
```

```
16   create componenttype Observer {
17     implements IObserver as iobserver;
18     uses ISubject as isubject;
19   }
20
21   create componenttype Subject {
22     implements ISubject as isubject;
23     uses IObserver as iobserver;
24   }
25
26   create connectortype Simple {
27     mode one2one;
28   }
29
30   create connectortype Aggregation {
31     mode one2many;
32   }
33
34   create linkagetype from Subject.iobserver to Observer.iobserver with Aggregation;
35   create linkagetype from Observer.isubject to Subject.isubject with Simple;
36 }
```

Listing 5.20: DAD-T model of the Observer pattern

The pattern is directly expressed as transformation rules in order to be injected quite easily. The **DAD-T** set is related to the specification of the Observer pattern in an **ASR** model where its benefits and limitations can be argued. Architects will need to write *mapping* rules, i.e. replacements, to identify the element(s) that must implement the Observer and the one that will be the Subject. Also, new connections and/or instances may be specified to define an **Assemblage** and/or **Deployment**. Listing 5.21 illustrates how such mapping rules will look like on an existing Target ComponentType. Only the Subject is merged into the Target's existing definition.

```
1  package be.iodass.example;
2
3  asrmodel be.iodass.example.dummyasr;
4  dadmodel be.iodass.example.dummydad;
5
6  transformationset injectobserver concerns Observer {
7    // include all rules from observer_pattern DAD-T set
8    include iodass.base.patterns.observer;
9    // replace the newly created Subject by the existing Target
10   replace Subject by Target merge;
11 }
```

Listing 5.21: Inject Observer pattern into a DAD model

The resulting model will look like the Listing 5.22. The Target received the Subject's Facets and the pattern LinkageTypes have been updated accordingly to the replacement transformation.

```
1  package be.iodass.example;
2
3  dadmodel dummy {
4    definition {
5      interface IObserver { /* omitting definition */ }
6      interface ISubject { /* omitting definition */ }
7      componenttype Target { // now uses the IObserver and implements the ISubject
8        /* previously defined elements stays untouched */
```

```
9       implements ISubject as isubject;
10      uses IObserver as iobserver;
11    }
12    componenttype Observer { // Observer is now part of model (no replace/merge)
13      implements IObserver as iobserver;
14      uses ISubject as isubject;
15    }
16    // new connectors included
17    connectortype Multi { /* omitting definition */ }
18    connectortype Simple { /* omitting definition */ }
19    // linkage types created with Target as part of them
20    linkagetype from Target.iobserver to Observer.iobserver with Multi;
21    linkagetype from Observer.isubject to Target.isubject with Simple;
22
23    /* omitting remaining of file */
24  }
25 }
```

Listing 5.22: Sample **DAD** model after injection of `Observer` pattern

Modelers can build their own list of architectural patterns, always related to their descriptions as **ASR** models. The amount of mapping rules is usually lesser to the number of model elements in the patterns, so remains quite low. The aforementioned `replacement` transformation mechanism updates the references of the introduced pattern elements to point to the target elements from the existing model. This update mechanism lower significantly the manual maintenance effort on the **DAD** model itself since no manual modification must be done to *implement* the connections between pattern elements that are merged into existing elements, thanks to `replacement` rules.

5.6 Inject Structural Changes into the Online Library

In the present chapter, we introduced and formalized the available transformation rules to be applied on **DAD** models in order to inject new requirements or maintain architectural models. We will now concretely illustrate the usage of these transformation rules onto our online library system, from a simple update of the model, to a more intrusive pattern injection. By these examples, we will highlight the benefits of such a formalized approach in terms of manual model maintenance and architectural changes traceability.

5.6.1 A First Example with a Simple Modification

As a first illustration, we introduce a simple evolution of the online library system. We suppose that the `Customer` may accept or refuse the book at the delivery. In any case, the `OnlineLibrary` will receive an acknowledgment message from the ParcelDelivery. If the book is refused by the `Customer`, then a credit note must be issued. We summarize the newly created **Asr** in Figure 5.3.

The complete **xEARS**-compliant descriptions of the new requirements are presented in the **ASR** model excerpt in Listing 5.23.

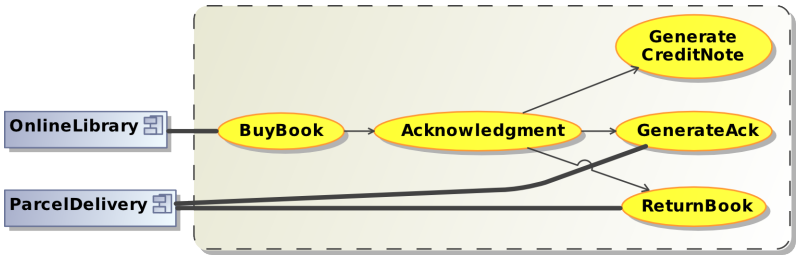


Figure 5.3: Graphical excerpt of the ASR model with the acknowledgment-related requirements

```
1  package be.iodass.onlinelibrary;
2
3  asrmodel onlinelibrary with be.iodass.onlinelibrary.onlinelibrary {
4    /* skip beginning of file with all other requirements that remains untouched */
5
6    func Acknowledgment assigned ParcelDelivery {
7      description "When the book is delivered, the ParcelDelivery shall acknowledge
8                   the reception of the book by the Customer to the OnlineLibrary. ";
9      refines BuyBook {
10       assessment "new requirement asking for an ACK at delivery.";
11     }
12   }
13   func GenerateAck assigned ParcelDelivery {
14     description "The ParcelDelivery shall generate an acknowledgment saying if the
15                  Customer accepted the delivery or not to the OnlineLibrary.";
16     refines Acknowledgment {
17       assessment "Must create ack message with customer and book references.";
18     }
19     realisation be.iodass.onlinelibrary.handleack {
20         assessment "Create needed ack-related interface since existing interface
21                  between both components hasn't the right polarity.";
22     }
23   }
24   func CreateCreditNote assigned OnlineLibrary {
25     description "When a delivery acknowledgment concern a refusal, the OnlineLibrary
26                  shall send a paper-based credit note equal to the book's selling price
27                  to the Customer.";
28     refines Acknowledgment {
29       assessment "Must refund customer.";
30     }
31     realisation be.iodass.onlinelibrary.handleack {
32         assessment "Create event in new interface to warn a refusal has been received";
33     }
34   }
35   func ReturnBook assigned ParcelDelivery {
36     description "When a Customer refuses a book, the ParcelDelivery shall send the book
37                  back to the BookStore.";
38     refines Acknowledgment {
39       assessment "A new delivery must be created in the system, system will have to
40                  listen for a new event.";
41     }
42     realisation be.iodass.onlinelibrary.handleack {
43       assessment "Warn itself by an event to generate delivery.";
44     }
45   }
46 }
```

Listing 5.23: **ASR** model excerpt for the acknowledgment requirement

Four new requirements are introduced in the existing **ASR** model (we only reproduced the concerned `Asr` in the above listing): one higher-level requirement refining the *"BuyBook"* `Asr` that generally describes the acknowledgment mechanism and three lower-level ones that detail the needed features to implement the new requirement. All `Asr` are *realized* by the same transformations set, which is presented in Listing 5.24.

```
package be.iodass.onlinelibrary;

asrmodel be.iodass.onlinelibrary.onlinelibrary;
dadmodel be.iodass.onlinelibrary.onlinelibrary;

transformationset handleack concerns
            GenerateAck, CreateCreditNote, ReturnBook {

  // generate ACK of delivery
  create interface DeliveryAcknowledgment {
    async bookDelivered(DeliveryDetails details, boolean ack);
  }

  alter componenttype ParcelDelivery {
    uses DeliveryAcknowledgment as dack;
  }

  alter componenttype OnlineLibrary {
    implements DeliveryAcknowledgment as dack;
  }

  create linkagetype from ParcelDelivery.dack to OnlineLibrary.dack with One2one;

  // credit note creation
  alter interface DeliveryAcknowledgment {
    add event generateCreditNote(DeliveryDetails details);
  }

  // return book to store
  create interface BookRefusal {
    sync boolean makeNewDelivery(DeliveryDetails details);
  }

  alter componenttype ParcelDelivery {
    implements BookRefusal as bk_impl;
    uses BookRefusal as bk_use;
  }

  create linkagetype from ParcelDelivery.bk_use to ParcelDelivery.bk_impl with One2one;
}
```

Listing 5.24: **DAD-T** set to inject the acknowledgment-related requirements

In this `DAD-T` set, we decided to explicitly separate each new concern with dedicated rules. The resulting model is then more verbose, but this design choice was made for demonstration purposes in order to identify the transformations related to each of the implemented requirement.

A condensed version of the resulting **DAD** model is reproduced in Listing 5.25.

```
package be.iodass.onlinelibrary;

dadmodel onlinelibrary {
  definition {
    /* skip unchanged data structures and interfaces */
```

```
6
7     componenttype Customer { /* unchanged */ }
8
9     // specification of the OnlineLibrary
10    componenttype OnlineLibrary {
11      implements BookSelling as bs;
12      uses BookCatalog as bc;
13      uses Auction as a;
14      uses Delivery as d;
15      // new facet created
16      implements DeliveryAcknowledgment as dack;
17    }
18    componenttype Bookstore { /* unchanged */ }
19
20    // specification of the deliverer
21    componenttype ParcelDelivery {
22      implements Delivery as d;
23      // new facets created
24      uses DeliveryAcknowledgment as dack ;
25      implements BookRefusal as bk_impl ;
26      uses BookRefusal as bk_use ;
27    }
28
29    connectortype One2many { mode one2many; }
30    connectortype One2one { mode one2one; }
31
32    // newly created interfaces
33    interface DeliveryAcknowledgment {
34      async bookDelivered ( DeliveryDetails details, boolean ack ) ;
35      event generateCreditNote ( DeliveryDetails details ) ;
36    }
37    interface BookRefusal {
38      sync boolean makeNewDelivery ( DeliveryDetails details ) ;
39    }
40    /* skipping untouched linkagetypes */
41
42    // new linkagetypes
43    linkagetype from ParcelDelivery.dack to OnlineLibrary.dack with One2one ;
44    linkagetype from ParcelDelivery.bk_use to ParcelDelivery.bk_impl with One2one;
45  }
46 }
```

Listing 5.25: Resulting **DAD** model after acknowledgment-related transformations

All transformation rules have been executed by the engine and the new `Interfaces`, `Facets` and `LinkageTypes` have been added into the **DAD** model.

5.6.2 Pattern Definition and Injection

As a second evolution of the online library system, we will transfer the auction responsibility from the `OnlineLibrary` to the `Bookstore`. Instead of having the library that leads the overall process, it will contact one of its bookstores (randomly chosen), and that store will lead the process until one winner is found. The winner will then contact the library to make itself known. We record these new requirements in Listing 5.26 (only the `Bookstore`-related ASR are reproduced).

```
1 package be.iodass.onlinelibrary [ revision 1 ] ;
2
3 asrmodel onlinelibrary_cb with be.iodass.onlinelibrary.onlinelibrary {
4
5   func DecentralizedAuction assigned Bookstore {
```

```
    description "When the BookStore is contacted by the OnlineLibrary to determine the
                 price of a book, the BookStore shall contact all other BookStores to
                 run an auction.";
    refines FindCheapestByAuction {
      assessment "Transfer responsibility from library to stores.";
    }
    realisation be.iodass.onlinelibrary.injectcallback {
      assessment "Must create a new interface to manage communication between stores.";
    }
  }

  func WonAuction assigned Bookstore {
    description "When the BookStore win the auction with the lowest price for a book,
                 the BookStore shall send its price and ID to the OnlineLibrary.";
    refines FindCheapestByAuction {
      assessment "Callback after auction win.";
    }
    realisation be.iodass.onlinelibrary.injectcallback {
      assessment "Use callback pattern.";
    }
  }
}
```

Listing 5.26: **ASR** model excerpt for the transfer of the auction leading

In the above code snippet, the *callback* pattern is required to be injected to implement the new requirements. The synchronized `getPriceForBook` service from the `Auction` `Interface` will be replaced by the *callback* pattern. Previously, this `Interface` was calling all `Bookstores` to collect their new offers lesser than a `currentprice`. Listing 5.27 specify the callback pattern as a **DAD-T** set[14].

```
package iodass.base.patterns;

import iodass.base.basic.constructs;
asrmodel iodass.base.patterns.callback;

transformationset callback concerns UseCallback {

  create interface Main { async call(); }
  create interface CallBack { async callback(); }

  create componenttype Caller {
    uses Main as main;
    implements CallBack as cback;
  }
  create componenttype CallerBack {
    implements Main as main;
    uses CallBack as cback;
  }

  create linkagetype from Caller.main to CallerBack.main with One2one;
  create linkagetype from CallerBack.cback to Caller.cback with One2one;
}
```

Listing 5.27: **DAD-T** set of the callback pattern

Now, we have to write the mapping rules from the pattern to the architectural elements and apply the remaining transformations to fully implement the aforementioned new requirements, as illustrated in Listing 5.28. After running the transforma-

[14] The `One2one` connector has been defined in a library model, pointed as `iodass.base.basic.constructs`.

tion that creates the pattern, we have to replace both the `Caller` and `CallerBack` pattern `ComponentTypes`, and override their `main` `Facets`. The pattern created a `LinkageType` on these `main` `Facets`, but a `LinkageType` on their overridden `Facets` already exist, so this creation rule will be implicitly removed. We map the `Main` `Interface` and alter its definition to transform the concerned `Service` into an *asynchronous* call. The concrete new `Interface` that will handle the callback also created and substituted to the abstract `CallBack` `Interface`.

```
1  package be.iodass.onlinelibrary  [ revision 1 ] ;
2  asrmodel be.iodass.onlinelibrary.onlinelibrary;
3  dadmodel be.iodass.onlinelibrary.onlinelibrary;
4  transformationset injectcallback concerns DecentralizedAuction, WonAuction {
5
6    // import pattern
7    include iodass.base.patterns.callback;
8
9    // map both ComponentTypes
10   replace Caller by OnlineLibrary merge overrides {
11     facet Caller.main by OnlineLibrary.a;
12   };
13   replace CallerBack by Bookstore merge overrides {
14     facet CallerBack.main by Bookstore.a;
15   };
16
17   // replace Main interface
18   replace Main by Auction;
19
20   // alter getPriceForBook
21   alter interface Auction {
22     rewrite Auction.getPriceForBook by async getPriceForBook(int isbn, float
           currentprice);
23   }
24
25   // create concrete callback interface
26   create interface AuctionCallback {
27     sync void sendPriceForBook(in int isbn, in float bestprice);
28   }
29
30   // replace abstract callback by concrete one
31   replace CallBack by AuctionCallback;
32 }
```

Listing 5.28: **DAD-T** set to inject the callback pattern and implement the new auction

After having run the transformation engine onto the **DAD** architectural model, the `AuctionCallback` `Interface` will encompass the callback service. The new `LinkageType` is also correctly created and rebound to the concrete **DAD** element, instead of their original abstract objects. Listing 5.29 presents the excerpt of the impacted **DAD** elements.

```
1  package be.iodass.onlinelibrary;
2  dadmodel onlinelibrary {
3    definition {
4      /* skip unchanged data structures and interfaces */
5
6      componenttype Customer { /* unchanged */ }
7
8      // specification of the OnlineLibrary
9      componenttype OnlineLibrary {
10       implements BookSelling as bs;
11       uses BookCatalog as bc;
```

```
12        uses Auction as a;
13        uses Delivery as d;
14        implements DeliveryAcknowledgment as dack;
15        // new callback Facet
16        implements AuctionCallback as cback ;
17      }
18      componenttype Bookstore {
19        implements BookCatalog as bc;
20        implements Auction as a;
21        uses AuctionCallback as cback;
22      }
23      componenttype ParcelDelivery { /* unchanged */ }
24
25      /* skip other untouched linktypes and interfaces */
26
27      // new interface for callback
28      interface AuctionCallback {
29        sync void sendPriceForBook ( in int isbn,
30        in float bestprice ) ;
31      }
32
33      /* skipping untouched linkage types */
34
35      // new linkage type for callback
36      linkagetype from Bookstore.cback to OnlineLibrary.cback with One2one ;
37    }
38  }
```

Listing 5.29: Resulting **DAD** model with the new auction

5.6.3 Concluding Remarks on the Examples

In both illustrations, the requirement changes have been documented within the **ASR** model and are clearly identifiable through transformation rules. The precise impacts on the architectural model is clearly known and reproducible, so that rollbacks to previous model versions are always possible. Also, the usage of design patterns is enhanced with straightforward injections requiring only a few mapping rules in existing **DAD** models. Furthermore, as for the callback pattern, the link between the call and callback services are kept as a model transformation (with an `override` rule), which enhances the traceability and documentation of the model.

5.7 Wrap-Up and Conclusions over DAD-Transformations

We introduced in this Chapter 5 our had-hoc transformation language for **DAD** architectural models. We first argued for a semi-structured documentation of design patterns in order to enhance their reusability across models and domains. We also debated about their role in system architecture maintenance, as well as their important role in knowledge transfer activities.

Considering model transformations as first class entities of architecture model design and evolution, we specified a concrete-syntax-based transformation language intended to manipulate **DAD** models in a formal and structured way. To this end, we introduced a set of transformation rules and formalized their semantics, when necessary. We also detailed the validation rules that are executed by the

transformation engine to avoid as many model violations as possible, or to support modelers with appropriate feedback. Because patterns play a key role in architecture design, the **DAD-Transformation** language offers dedicated means to specify and inject patterns into **DAD** models with few coding effort. However, only the structural part of patterns is definable in a **DAD-T** set, the justifications and explanations being defined in the linked **ASR** model. It is then highly possible that all existing patterns may not be *automated* as a set of transformation rules.

Still, the additional work required to write transformations instead of directly editing the model is not negligible. The amount of lines of code required by the first modification in the illustration is higher than the number of lines produced in the resulting model. At the opposite, for pattern injections, modelers are required to write mapping rules between the pattern elements and their targets in the existing model and the subsequent updates are performed by the transformation engine, such that the coding effort is lowered.

Even if this transformation-oriented mechanism is not trivial, we argue in the current thesis that this supplementary effort is more than equally balanced by the improvements in architectural maintenance, evolution and knowledge management, as we will detail in the next two chapters.

CHAPTER 6

PUTTING IT ALL TOGETHER, THE IODASS FRAMEWORK

This chapter is founded on the three preceding languages and introduce a methodological framework for software architecture design and evolution. We first depict an iterative and transformation-oriented design method that rely on requirement explorations and model transformations. Second, we evaluate our proposal regarding a general model built on common concepts from academic and industrial architecture design methods. Finally, we introduce the tool support for the modeling languages introduced in this dissertation, as well as the underlying tool frameworks.

6.1 Need for more Agility and Traceability in Software Development

As we already stressed many times and especially in Chapter 1, software systems must evolve to stick to changes in requirements made by the stakeholders. In order to keep control over a system, architectural modifications should be recorded together with the descriptions of their requirements and design rationale.

Iterative design methods are based on step by step development of a software system by integrating iteratively new concerns and/or fixing bugs in preceding work or prototypes [Gilb, 1981; Boehm, 1986; Beedle et al., 1999; Larman and Basili, 2003]. Many of these methods use a formalized document to pinpoint the objectives for the

next iteration and evaluate the developed prototype or incomplete product regarding these objectives. The traceability between iterations is therefore guaranteed, at the moment both the analysts and the developers follow the iteration or release plan.

However, the traceability during an iteration is harder to ensure, because either the tool support is lacking, not appropriate or because the perceived return on investment is too small [Ali Babar et al., 2006; Bjørnson and Dingsøyr, 2008]. As we detailed in Section 1.3, many tools and methods have been created to tackle that problem, developed in the academia as well as in the industry, and a couple of them have been evaluated on industrial projects to capture design rationale [Ali Babar et al., 2008; Bracewell et al., 2009; van Heesch et al., 2013].

In all these tools and methods, documentation activities must be performed aside design activities. Design decisions and implementation alternatives must be recorded explicitly in a dedicated tool such that a significant extra work must be performed by the designers. In the present chapter, we introduce a methodological framework that partially reduce the needed extra effort to document and trace explored alternatives and design decisions through model transformations.

6.2 Pick One, Document and Transform

On top of our three specific languages we depicted in the preceding chapters, we introduce an iterative architecture design cycle, called **IODASS** (p**I**ck **O**ne, **D**ocument **A**nd tran**S**form **S**trategy). The overall idea is depicted in Figure 6.1.

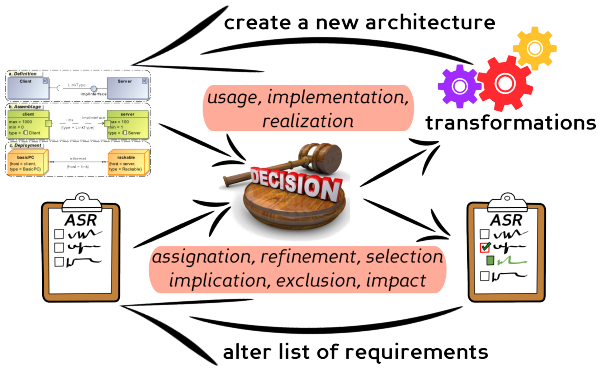


Figure 6.1: Overview of **IODASS** transformation method framework

As starting points, architects need a first architectural representation expressed as a **Definition-Assemblage-Deployment** (**DAD**) model and a first list of **Architecturally Significant Requirements** (**ASR**). Both models may be very naive and/or incomplete. For example the **DAD** model may contain a single `ComponentType` to which all requirements are assigned.

From these models, they iteratively make decisions in the requirement listing that may also affect the architectural model. For each decision, they record it directly in the **ASR** model with its rationale, as presented in Chapter 4. In case architectural

changes are needed, they will also write a set of model transformations to implement their decision into the architecture representation.

Anytime in the design process, architects are able to come back to a previously defined architecture model to take other decisions and build alternative architecture representations. To this end, instead of directly modifying the **DAD** model, the proposed framework keeps a history of previous models. The complete decision paths and model transformations, *i.e.* `Realization` **DAD-T** sets, that lead to a particular architecture are stored into a *revision-tree*. This mechanism is analogous to *Source Code Management* (SCM) systems, like *Subversion*[1] or *Git*[2]. We illustrate a fictitious **IODASS** revision-tree in Figure 6.2.

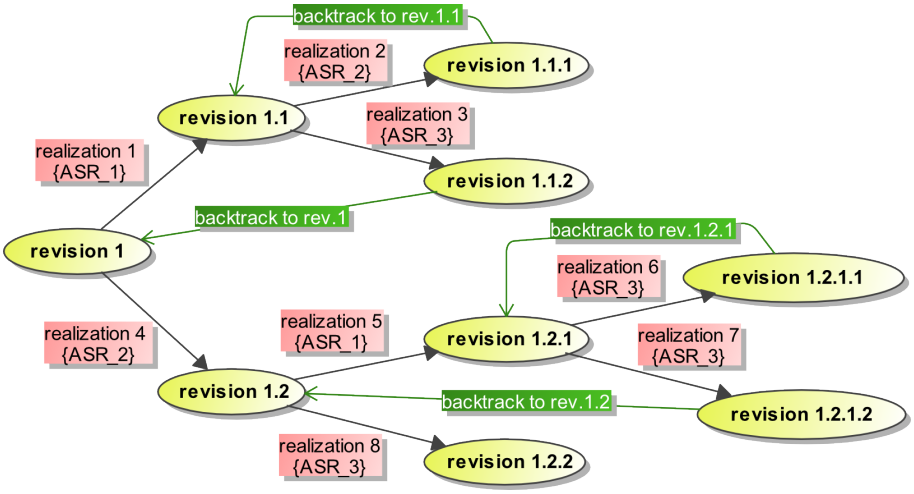


Figure 6.2: Overview of a fictitious **IODASS** revision tree

By convention, we decided to start from *"revision 1"* as the first (**DAD**,**ASR**) pairwise models. Every `realization` decision will generate a new revision, suffixed with *".1"*. New branches, on the other side, are numbered by adding *1* to the highest revision number of the same level. For example, in the above revision-tree, from *"revision 1.1.1"*, we backtracked to *"revision 1.1"* to create the new branch numbered *"1.1.2"*. The edges in the above figure are also labeled with the `realization` sequence numbers and the related **ASR**. For example, we first built an architecture model by implementing the `ASR_1` from `decision1`, but finally backtracked to the initial *"revision 1"* to start over by implementing first the `ASR_2`, and so forth.

Formally, the **IODASS** iteration loop can be expressed as the UML activity diagram illustrated in Figure 6.3. Prior to a **IODASS** iteration, architects must have written a first version of both a **DAD** and **ASR** models. The process is repeated until no more requirements (`ASR`) must be taken into account in the architectural model. For each iteration loop, a particular revision number is selected as the *base* revision under

[1] https://subversion.apache.org
[2] http://www.git-scm.com

development. From this revision, one ASR is selected on which some decision is made and its rationale is documented directly into the current ASR model. If needed, *i.e.*, a `Realization` decision has been made, some transformation rules must be specified into a **DAD-T** set. These transformations are then performed onto the current **DAD** model. The resulting **DAD** model is finally reviewed and evaluated, depending on the modelers' criteria and method.

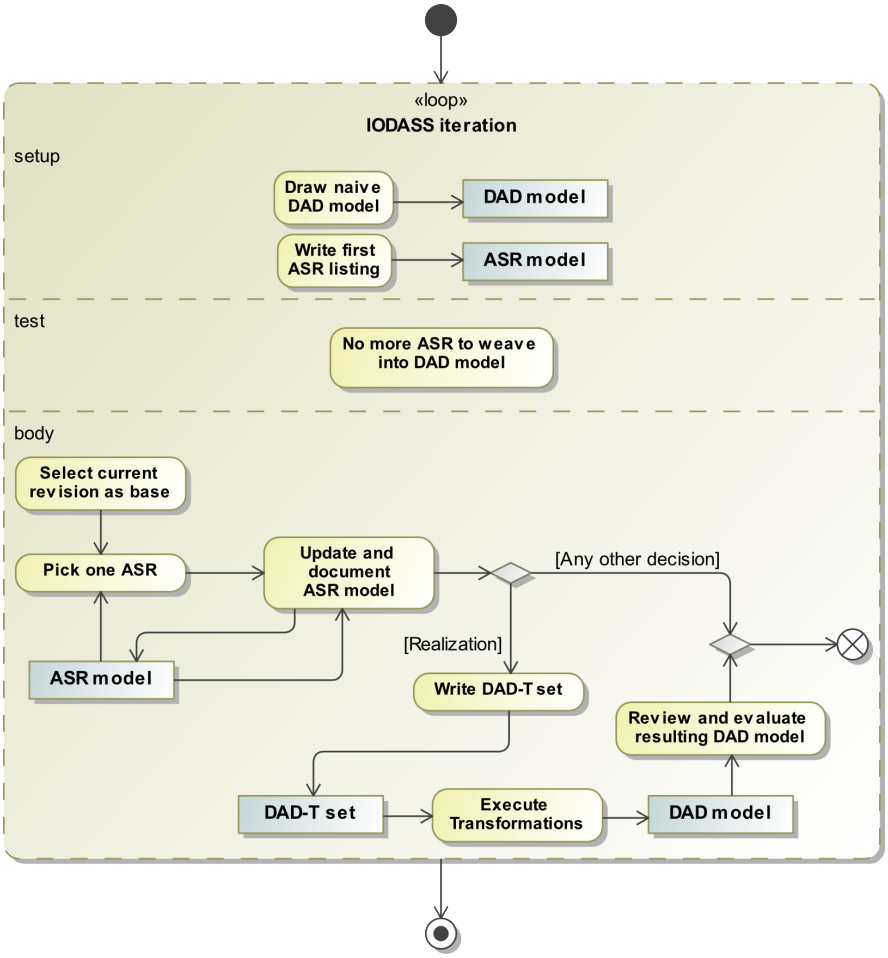


Figure 6.3: **IODASS** iteration expressed as a UML activity diagram

In our design cycle, we do not fully consider evaluation or *prioritization* activities. The cycle focuses on the recording of architectural design decisions only. The proposed method may be combined with other methodological frameworks dedicated to requirement engineering for prioritization or dedicated architectural evaluation techniques. Modelers are then free to use whatever method or standard for these two aspects. In the following section, we confront our proposal with a theoretical evaluation grid for architecture design methods.

6.3 Evaluation Regarding the General Model of Software Architecture Design

From five industrial architecture design methods, Hofmeister *et al.* specified *« a general model of software architecture design »* [Hofmeister et al., 2007]. The authors analyzed the *Attribute-Driven Design* [Bass et al., 2003; Wojcik et al., 2006], *Siemens' 4 Views* [Hofmeister et al., 1999], *Rational Unified Process®4+1* [Kruchten, 1995; Kroll and Kruchten, 2003], *Business Architecture Process and Organization* [Obbink et al., 2000; America et al., 2004], and *Architectural Separation of Concerns* [Ran, 2000] method, and identified their commonalities.

Figure 6.4 summarizes the activities and artifacts of their *"ideal pattern"* that we will use to evaluate our design strategy[3].

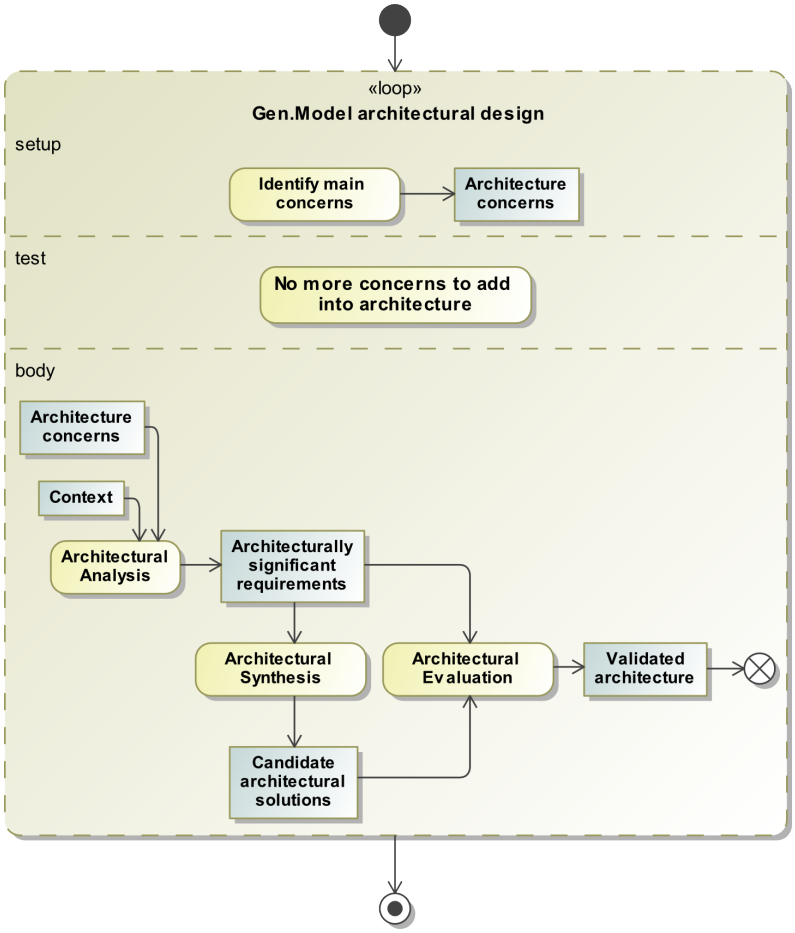


Figure 6.4: Architectural design-related activities as defined in the general model

[3]The original process specified in [Hofmeister et al., 2007] has been reformulated as a UML activity.

During *Architectural Analysis*, *Architecture concerns* are evaluated and reformulated into *Architecturally significant requirements*, using the *Context, i.e.*, the developmental, operational or political environment that may influence the system's development [IEEE, 2000]. The produced list of ASRs is either used to propose a solution during the *Architectural Synthesis* and some more *Architectural Evaluation* must be performed to ensure the decisions made until now are right. The *Candidate architectural solutions* are also confronted to the expected ASRs, when created from the *Synthesis* activity, to produce a *Validated architecture*. Both *candidate* and *validated architectures* must be accompanied by their design rationale.

This overall iterative process is influenced by the architects' knowledge over *Design*, *Analysis* and *Realization*. *Design knowledge* encompasses every structured and unstructured knowledge regarding how to build an architecture. *Analysis knowledge* concerns analytic methods depicted to evaluate an architecture regarding some criteria. *Realization* knowledge focuses on project management or technological solutions to support the development of a software architecture.

Analogously to what we identified in Section 1.1, the general model relies on an iterative design method matching the *« grow, don't build software »*'s view [Brooks, 1987]. The authors gathered in the concept of *backlog* the list of explicit documents with ASRs, constraints, ideas or issues concerning the current development of the software, as well as the ordering, relations and priority between the architectural concerns. *Backlog items* are constantly moving, being added or removed from the list such that at any time, the backlog can give an updated snapshot of the current development. This backlog is close to the *product backlog* as defined in the SCRUM method [Schwaber and Beedle, 2001].

From Figure 6.3 and 6.4, we can identify many commonalities in both development cycles. This is no surprise since the general model has been defined as a *pattern* architecture design based on industrial methods and our cycle is highly iterative too. For all activities and outputs depicted in Figure 6.4, we detail their counterparts in the **IODASS** method.

analysis The analysis is directly performed in the **ASR** model where modelers may take decisions over requirements and record the design rationale.

synthesis The synthesis is performed through model transformations, by successively integrating new `Asr` into the **DAD** model.

evaluation After injecting a transformation set, modelers may review the produced model and continue with this new model or backtrack to a previous revision.

concern A concern is always addressed in terms of architecturally significant requirements. Broader concerns or the ones that are not directly linked to a system architecture are not taken into account in the **IODASS** strategy.

context Likewise concerns, the context that has no direct impact on the architecture is not addressed; contextual elements that have a meaning for the architecture may be expressed as `Asr`'s `Rationale`, like `Constraints` or `Assumptions`, for example.

ASR The `ASR` is the central point of the **IODASS** strategy and is expressed

by a dedicated construct.

candidate Any transformed model is a candidate architecture until it is not yet reviewed; every candidate is kept in the revision tree.

validated The validated architecture can be identified as the latest leaf created in the revision tree; no specific tag is defined to highlight a given model as validated. However, `ComponentTypes` may be tagged as *final*, such that their definitions may not be modified by subsequent transformations in the same revision path.

backlog Coupled to the definition of `Asr`, priority attributes may be added into the model; any other methodological attribute may be expressed either as user-defined properties, or using the dedicated `Rationale` constructs.

The **IODASS** methodological framework addresses all activities as well as all generic artifacts, except for the concerns and context where our method focuses only on the architecture-related concepts. Project management-related characteristics must be first encoded as user-defined properties in a dedicated **DAD** property model before being used. This mechanism is flexible enough to add a wide range of properties, but requires modelers to define them prior to design activities.

6.4 Tool Environment for IODASS Languages

In this Section, we will introduce the tool support for all **IODASS** languages. First, we discuss the available alternatives that we evaluated as a basis for the envisioned tool suite. Second, we present the underlying technological frameworks we reused to build our tool. Finally, we detail the set of tools we developed[4].

6.4.1 Preliminary decisions

When considering a transformation-oriented design method, the question of the tool support arose quickly and multiple alternatives were evaluated. We summarize here the main options we considered with their benefits and disadvantages.

Build everything from scratch

A first option was the possibility to build a completely new software from scratch. But this option requires to write first a *parser* and a *lexical analyzer*, also called *lexer*, for all languages in order to *read* a model, syntactically validate it and manipulate it through model transformations. Writing its own parser and lexer would produce more user-friendly error messages and would probably stick closer to the *"ideal"* syntax as seen by the language creator. However, many generators exist in the market and building everything from scratch would have needed a subsequent development and testing time. Since we were in a research context with languages

[4] A user guide is available in Appendix E with a more complete description of all features as well as some detailed implementation aspects.

subject to syntactical and semantical changes during software development, this option has been quickly dismissed.

Parser and lexical analyzer generators

Many lexical analyzer generators have been developed for many target programming languages, such as Java, C, C#, JavaScript, and so forth. These generators usually reduce the needed development time and ease the maintenance since language evolutions are almost always limited to *"update the grammar and press the generate button"*. A couple of generators are also integrated into larger development toolkits with textual-based editors. However, depending on the grammar language, the DSL grammar can contain some *tweaks* to be accepted. For example, some parser generators may accept left recursion or not, which influence the way a grammar is defined as well as its accepted syntactical constructions. But, again because we were developing the tool support at the same time as the design strategy, an automated generator was a more practical solution, even if we needed to define a *"tool-compliant"* grammar. Also, because some lexer generators are well integrated with textual editor generators in larger DSL frameworks, this option was selected.

Build upon an existing DSL definition framework

Recently, some meta-modeling and Domain Specific Modeling Language (DSML) frameworks have appeared here and there, like JetBrains Meta Programming System[5], MetaEdit+[6], MetaDONE[7] or Xtext[8]. Those modeling facilities have the advantages to automate the generation and maintenance of DSL-based environments. When such tools are interfaced with an *Integrated Development Environment* (IDE) such as Eclipse[9], Microsoft® Visual Studio[10] or to a lesser extent, SimuLink[11], modelers may benefit from other *compatible* tools and create interfaces with them. Because of these integration and extendability opportunities, we favored a DSML framework integrated into an existing IDE.

Reuse existing transformation language

Model transformations play a key role in the **IODASS** architecture design strategy. We widely discussed our view in Section 1.4 and Chapter 5 where we argued for a concrete syntax-based transformation language. We evaluated the possibility to reuse an existing transformation language as the *back-end* implementation for our method. Again, many transformation languages have been implemented into IDEs,

[5] http://www.jetbrains.com/mps/
[6] http://www.metacase.com/
[7] http://www.metadone.be
[8] https://www.eclipse.org/Xtext/
[9] http://www.eclipse.org
[10] http://www.microsoftstore.com/Visual-Studio/
[11] http://www.mathworks.nl/products/simulink/

such that they are often able to manipulate models expressed in a *standardized* common language. However, for this particular aspect, we wanted to be as independent as possible to any existing transformation facility for the following reasons.

Since we decided to go for a concrete syntax-based transformation language, we would have needed to write a code generator to translate **DAD-T** rules into that intermediate language. Similarly to the parser-generator that somehow constraints the syntax of a concrete language with its own grammar, the target transformation language would have constraint the concrete syntax of the **DAD-T** rules, and possibly the semantics in some scenarios too. Moreover, the `replacement` and `deletion` policies would have been complex to handle since more transformation rules must be created too, so possibly more debugging and code maintenance. Many transformation languages are also evolving syntacticly and semantically. Many of them are products coming from the academia and the backward compatibility is not always ensured. For all these reasons, we preferred to stay independent from any existing model transformation framework.

Summary table

We summarize in Table 6.1 the main intended assets and their evaluated positive or negative impacts as tooling solutions.

OPTION	EDITION	PARSER	LEXER	TRANSFO.	UPGRADE	DEV.TIME
From scratch	–	–	–	–	+	– –
Gen. parser/lexer	–	+	+	–	+	–
Existing framework	+	+	+	–	+	+
Exist. fk & transfo	+	+	+	+	–	+

Table 6.1: Summary of preliminary decisions

Regarding the above discussion, the very last choice we had to make concerned the selected DSML framework. The Eclipse IDE somehow imposed itself for many reasons. It provides a modeling framework completed with many extensions for visualization or manipulation purposes. For example, graphical editors can be created to edit custom models, or to generate code from models. It is free of charge and the source code is under a *copyleft* license[12]. Many Eclipse extensions are also provided under a copyleft license, which eases the development of custom tools and enhances code reusability. Many parser generators are also available as Eclipse plugins, built on top of the modeling framework for some of them. Besides all these arguments, the Eclipse ecosystem relies on a very wide community of developers, individuals, research groups or even commercial companies, which enhances the support possibilities for developers.

[12] In brief, the copyleft is a special type of author's right where the user of a software has the access to the source code, may modify it or reuse it, at the moment he keeps the copyleft for the new product.

6.4.2 Basis for the IODASS tool, the Eclipse Modeling Framework

The *Eclipse Modeling Framework* (EMF)[13] is a platform designed to manipulate structured models expressed in the *XML Metadata Interchange* (XMI) format [OMG, 2013b]. EMF provides many tools to write (meta-)models, visualize them and produce Java code to manipulate custom DSL models defined using the framework. The origin of EMF traces back in the beginning of the 2000's and is completely integrated into the Eclipse system. Many extensions exist, for model validation or model exploration through relational queries, for example.

EMF relies on the Ecore language to specify meta-models. Ecore is an implementation of the *Essential Meta Object Facility* (EMOF) [OMG, 2014b], being a simplified version of MOF, more closely related to object-oriented programming languages concepts. The Ecore language offers sufficient facilities to express meta-classes with their attributes, operations and relationships. From a meta-model expressed in Ecore, EMF is able to generate simple editors and Java code.

6.4.3 Xtext, an Extensible Framework for Domain Specific Languages

On top of EMF, the Xtext framework is a tool that automates the creation of DSLs textual editors as Eclipse plugins. Typically, from an *Extended Backus Naur Form* [Wirth, 1977] (EBNF)-like grammar, Xtext generates an *EcoreModel* that represents the DSL meta-model[14]. That Ecore meta-model is used to translate a given *SemanticModel*, *i.e.* the DSL model, into an EMF-compliant representation. Together with this representation, the abstract *ParseTreeModel* nodes are created to ease its manipulation by other generated Xtext artifacts. The grammar rules are also transformed into a dedicated *GrammarModel* that contains the *Parser* and *LexerRules*.

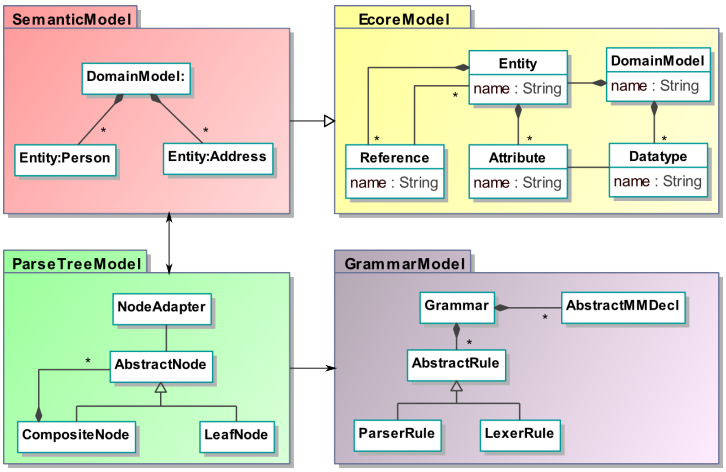


Figure 6.5: Overview of the Xtext data model (from Xtext documentation)

[13] http://www.eclipse.org/modeling/emf/

[14] Xtext can also generate a "template" grammar from an existing meta-model expressed in Ecore

As depicted in Figure 6.5 reproduced from the Xtext documentation, the *SemanticModel* must comply to the *EcoreModel* DSL meta-model and is manageable as a *ParseTreeModel*, which is its abstract syntax tree. This AST must respect the *GrammarModel* that specifies the acceptable grammar rules.

Xtext works on so-called *XtextResources* that are purely text files, *i.e.* the DSL textual model. As shown in Figure 6.6, a *Parser*, *Lexer* and *Serializer* are devoted to manipulate this textual model. These three tools are created by the Xtext framework, but they can be extended by DSL developers to tune some features or even they can be totally substituted, at the moment they respect the interfaces defined in the Xtext framework.

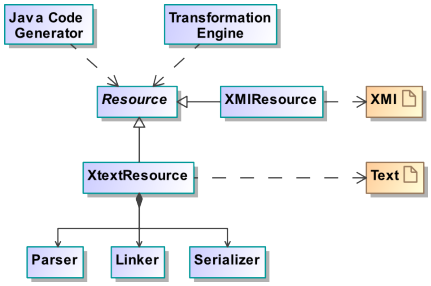


Figure 6.6: Overview of the Xtext framework (from Xtext documentation)

From an *XtextResource*, Xtext creates an EMF *Resource* that may be used to generate Java code or, like in our **IODASS** strategy, that can be *programmatically* transformed. Any *Resource* can be serialized as an XMI file, when transformed into an EMF *XMIResource*. This way, any EMF-compliant plugin may interact with Xtext-generated DSL editors because from a textual model that can be interpreted by an Xtext editor, an *XMIResource* can be created to be manipulated by any EMF-based Eclipse plugins. These interaction possibilities may be useful when building graphical visualizations or editors for Xtext-based DSLs.

At current time, more than 40 projects are referred from the Xtext DSL developers community. Some of them are even used in production environments, like the Eclipse OCL editor[15].

6.4.4 Support for Designers, the IODASS Textual Tool Suite

As proofs of concepts, we developed a set of textual editors as Eclipse plugins, built using the Xtext framework. All three languages presented in Chapters 3, 4 and 5 were implemented as separate plugins. The **DAD** property formalism was also implemented separately for independence and extendability reasons. The Xtext-compliant meta-model has been extracted as well in its own plugin for the same reasons. We finally came up with nine plugins, every language composed by two plugins of the form:

[15] http://projects.eclipse.org/projects/modeling.mdt.ocl

`iodass.textual.xx` the DSL model back-end
`iodass.textual.xx.ui` the textual editor

The ninth plugin contains the Ecore meta-models and the generated Java code to manipulate the languages objects, namely `iodass.model`. Typically, the model back-end contains classes and methods regarding name formatting, serialization, model elements scoping, and so forth. The textual editors focus on customization of the editors themselves, the possible verifications with their *quickfixes*, when possible, and other Eclipse-related customizations like project wizards. The transformation engine has been developed inside the **DAD-T** plugin, namely `iodass.textual.transfo`. A basic Java generator has also been developed, that creates template Java files from `DAD` models.

All grammars of the **IODASS** languages are reproduced at the end of this dissertation, in Appendices A to D. In the following, we will first introduce the editors, then we will concentrate on the transformation engine, and finally, on the basic Java templates generator.

Textual editors

As an example, Figure 6.7 shows a screenshot of the **IODASS DAD-T** editor within Eclipse.

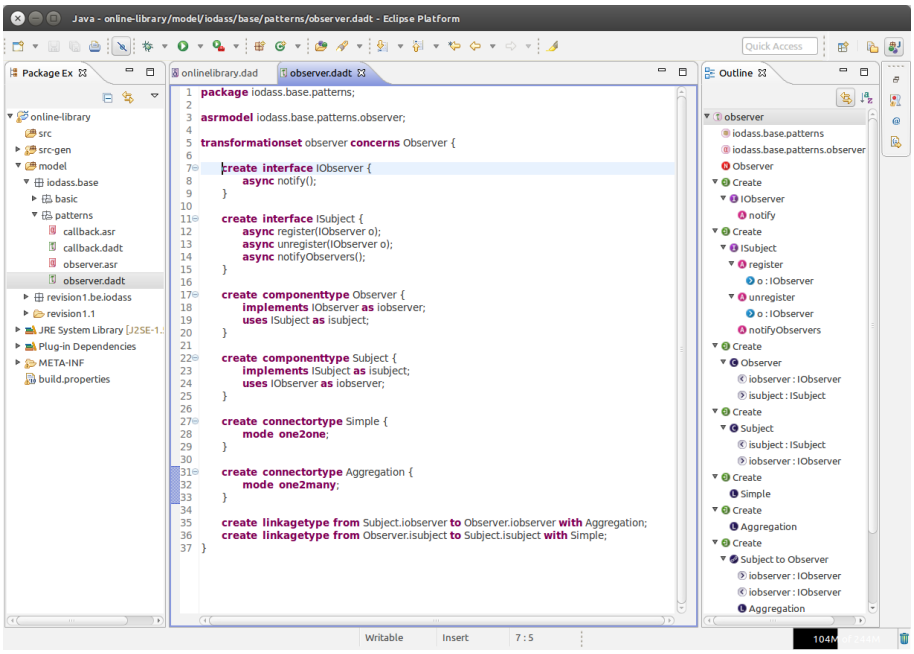


Figure 6.7: Overview of the **IODASS** tool suite

On the left hand side, the *Package Explorer* shows the **IODASS** revision packages, the `iodass.base` library and dedicated `src` and `src-gen` folders for the actual

software (generated) source code. In the middle of the frame, the source code of the current model is shown in the *Editor* part. On the right hand side, the *Outline View* shows a summary of the model elements present in the current model. This outline allows modelers to follow the referenced or imported model elements and navigate across models easily, just by clicking an element from the outline.

While editing models, a series of verifications are performed on the edited model, some of them being executed on file saving actions only to avoid greedy verifications to burden the performance of the editor. As presented in Section 3.2.7, the `LinkageTypes` are verified to ensure the polarities of the connected `Facets` are respected. Their compatibilities are also verified, as specified in our *duck typing*-like verifications. `Linkages` and `Plugs` are checked according the rules specified in Sections 3.3 and 3.4 as well. Many more verifications are executed regarding name uniqueness, actual presence of mandatory attributes and so forth.

When an error is found, the dedicated *Problems* view from Eclipse will contain a meaningful message with the details and the line number of the found problem. The line concerned by the problem will be underlined in the model source code too, as shown in Figure 6.8.

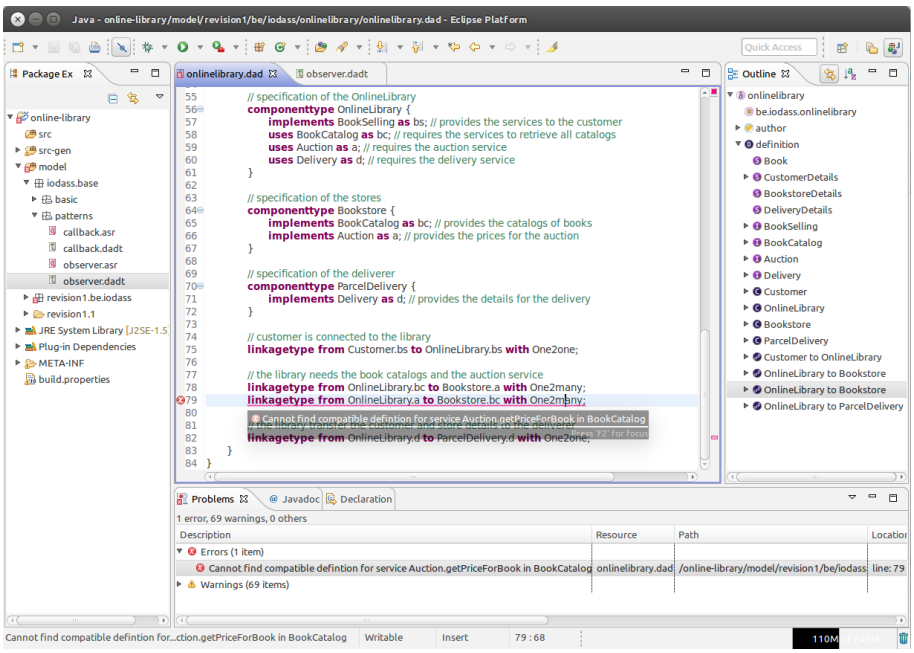


Figure 6.8: *Problems* view and meaningful error messages

In the above screenshot, the `OnlineLibrary.a` `Facet` has not the right polarity and both involved `Facets` have incompatible types. The errors concern the same `LinkageType` at line 75, which is underlined in red and displayed with a specific marker in the margins, just as for Java source code, or whatever programming language opened in a dedicated editor within Eclipse.

Transformation engine

The transformation engine has been developed in an aside language to Xtext, called Xtend[16]. It is a hybrid object-oriented and functional programming languages, close to the Scala [17] and Groovy [18] languages. Xtend classes are pre-compiled into pure Java code and its syntax is relatively close to Java, but with higher expressiveness. Xtend accepts, for example, lambda expressions and offers a powerful polymorphic dispatching facility to call methods depending on the object type at runtime.

The engine is callable from the contextual menu activated from a **DAD-T** model, in the *"IODASS"* menu. When a **DAD-T** set is run over a **DAD** model, a new *revision* folder is created, according to the naming conventions given earlier in Section 6.2.

The verifications explained in dedicated sections from Chapter 5 were also implemented with as much as possible meaningful messages. However, as already discussed in the aforementioned chapter, some errors may prevent a complete **DAD-T** set to be executed, or in some specific cases, a resulting model may contain some errors that must be handled manually.

Java code templates

A simple Java generator has been developed to create template Java source files from **DAD** models. `DataStructures` and `PrimitiveTypes` are translated into Java classes. `Interfaces` become Java interfaces containing the signatures of all their `Services`. Finally, `ComponentTypes` are generated as abstract Java classes that implement the *implemented* `Interfaces`.

All these Java sources are created into the `src-gen` folder of the current project, following the same package structure as the **DAD** model. The developers are able to fully implement the generated templates into the dedicated `src` folder. This separation, which is very common in MDE approaches, isolates the generated sources from the hand written code to be able to re-generate some models without overwriting existing code. However, depending on the structural changes applied by a modeler into a **DAD** model, some features in the hand written code may be broken if some significant changes have been applied to previously generated resources.

6.4.5 A Visual Notation for IODASS Models with MagicDraw

Concurrently to the textual editor, a simple custom *palette* has been defined in the MagicDraw[19] modeling tool. Part of the **DAD** and **ASR** language syntaxes have been specified as a UML profile and custom diagrams have been created in the tool. However, this graphical notation does not allow to represent all features of the languages and no connector exists between the textual and the graphical syntaxes.

This graphical notation has been mainly defined as a visualization mean to **DAD** models. Ideally, an EMF-compliant graphical notation should be developed in order

[16] http://www.eclipse.org/xtend/
[17] http://www.scala-lang.org
[18] http://groovy-lang.org
[19] http://www.nomagic.com/products/magicdraw

to benefit from both textual and graphical representations. An early prototype in the Eclipse's *Graphical Modeling Framework* (GMF)[20] has been defined, but quickly abandoned, mainly for maintenance, customizability and complexity reasons.

Lately, some promising graphical frameworks have appeared in the Eclipse ecosystem, like Zest[21] or Graphiti[22]. Those tools worth exploring, but we could not invest enough time in the graphical visualization to be able to build a complete tool fully integrated with the textual editors.

6.5 Wrap-Up and Conclusions Over The IODASS Framework

We depicted in this chapter a novel architecture design cycle named **IODASS** , *i.e.* the p**I**ck **O**ne, **D**ocument **A**nd tran**S**form **S**trategy. This design method is structured around the modeling languages we introduced in Chapters 3, 4 and 5. Its main objective is to encourage modelers to document their decisions, formally record *revision deltas* between models and keep traces of explored alternatives. We compared our method to a theoretical *"ideal pattern"* extracted from industrial and academic architectural design methods where, for each activity and document, we could provide a **IODASS** artifact or task.

We developed a proof-of-concept tool suite where we implemented our three languages and the model revision mechanism we introduced in the present chapter. We summarized the evaluated alternatives as underlying tool environment and we provided some details about its implementation. We already mentioned the limitations of purely textual editors, which may seem less intuitive to end-users, but that are usually more expressive and complete. As already argued in Chapter 4, the combination of both representation would be more suitable for architecture designers to have, on one hand a graphical big picture of the system and on the other hand, the full details in a declarative manner.

Having this transformation-wise software architecture framework in mind, we will now provide in Chapter 7 the details concerning the academic evaluation we conducted over students at the University of Namur. As we will detail in the following, the study was conceived with mainly two objectives: evaluate the approach regarding its feasibility and regarding the expressiveness of the modeling constructs we introduced.

[20] GMF is part of the Graphical Modeling Project, http://www.eclipse.org/modeling/gmp/
[21] http://www.eclipse.org/gef/zest/
[22] http://www.eclipse.org/gef/zest/

CHAPTER 7

CHALLENGING THE IODASS METHOD

This chapter presents the controlled experiment we conducted to evaluate the **IODASS** *framework on a class of students at the University of Namur. We first discuss about the alternatives we had and the expected outputs we can expect from the chosen evaluation strategy. Second, we detail the protocol we followed during the experiment and present the results from a quantitative and qualitative point of view. We finally analyze the results of the experiment and criticize the overall evaluation process, as well as its limitations.*

7.1 Evaluation Strategies and Their Outcomes

Now we have defined our architectural framework, we will evaluate its benefits and shortcomings in an empirical manner. Theoretically, many empirical alternatives exist. Wohlin *et al.* categorized them in three strategies [Wohlin et al., 2012]:

surveys introspective interviews and questionnaires
case studies empirical observations of monitored or retrospected projects
experiments laboratory experimentations under controlled conditions

In order to select the most appropriate strategy, we need first to question ourselves on the purpose of the study, *i.e.* what do we want to evaluate and how can we extract some valuable observations regarding what we want to evaluate. Second, the level of control we want to/may have on the study itself is also important. The level of control for a retrospective case study on a large-scale project conducted in the

industry is completely different from the one for a controlled experiment performed in an academic lab. Last, the cost and risks should be evaluated. For example, one may try to lower the cost for highly exploratory studies, since they might fail more frequently than confirmatory ones.

In a sense, a trade-off must be found between the available resources in terms of subject people and money to inject in the study and, probably the most critical, what exactly do we want to evaluate.

7.1.1 What Do We Want to Evaluate ?

Our transformation-centric framework is rather stringent. So, one of the most prominent aspect to evaluate was the feasibility of the general approach. We first had to ensure that an architecture model can effectively be created by successive formal transformations without requiring a colossal extra work for modelers.

As a second evaluation criteria, we had to identify whether the architectural language was sufficiently expressive to represent a software architecture. Likewise, we had to evaluate if the modeling elements we introduced were all useful. This was especially the case for the novel elements, mainly regarding the communication `Protocol` and `LinkTypes`.

Last, we decided to evaluate the benefits of our transformation-centric framework regarding an industrial modeling standard coupled to an iterative architecture design method. By comparing simultaneously the deliverables produced by *"software engineers"* using either one method or the other, we intended to analyze the advantages and drawbacks of our framework. As a side effect, we wanted to improve both our method and the modeling languages.

7.1.2 Available Resources and Their Impact on External Validity

For such a design method supported by a set of new DSLs, like for many other empirical evaluation in the software engineering field, experienced practitioners would be ideal subjects. They already have a (strong) background in design activities, they usually know about patterns, documentation and design rationale. A probably effective evaluation strategy would be to let them *play* with the languages on architecture modeling tasks in their day to day work on real world problems and then debrief with them in focus groups. Still, such an approach would require enough contacts in the industry with architects opened to be taught on our method and to do their job with our tools without being sure they could actually reuse it. Also, we needed a significant amount of participants to get sufficient feedback and being able to somehow *randomize* their profiles. However, we could not find any partnership that could provide us this level of guarantee.

As we are evolving in an academic world, a second source of *"engineers"* was more easily accessible, with a rather large amount of participants. During their two last years of Bachelor's degree, students at the University of Namur follow consequent modeling and object oriented courses that they have to practice in two team projects during their first year of their Master's degree. These projects require

them to model and implement software systems that combine web technologies, distributed communications and database management.

Höst *et al.* studied the differences between using professionals and students as study subjects to assess the impacts of ten lead-time factors to projects, like the competence, turnover, time pressure, etc [Höst et al., 2000]. In their comparative study, they observed that the differences between both groups were minor, but taking into account they used fourth year students that followed software engineering courses.

In a large survey on controlled experiments in software engineering published from 1993 and 2002, Sjøberg *et al.* identified that a very large proportion of those experiments were conducted on students [Sjøberg et al., 2005]. For such experiments, the question of the external validity, *i.e.* to what extent the results may be generalized to other persons, settings, causes and effects [Campbell and Stanley, 1966; Cook and Campbell, 1979; Shadish et al., 2001], is particularly important since the software engineering field usually targets the world of professionals.

In the remaining of this chapter, we will detail the protocol we followed to conduct our empirical study. We built our research on the many advices defined in recognized publications from Cook [Cook and Campbell, 1979], Pfleeger [Pfleeger, 1995] Carver *et al.* [Carver et al., 2010], and Wohlin *et al.* [Wohlin et al., 2012]. We also reproduce the results we observed during the study and analyze them within the goal/question/metric approach, as defined by Basili *et al.* [Basili, 1992; Basili et al., 1994]. To complement our observations, we submitted a questionnaire-based survey to all participants that we will discuss in this chapter too.

7.2 Protocol of the Comparative Case Study

Based on the observations we detailed in the previous section, we set up a comparative case study around these four questions, the two last ones addressing the third general goal of Section 7.1.1 :

(1) Is a transformation-centric approach feasible to build a software system from scratch?
(2) What are the benefits of a transformation-centric approach to handle architectural evolutions?
(3) How expressive is the set of **IODASS** languages to represent software architectural models?
(4) How expressive is the set of **IODASS** languages to trace architectural design decisions and rationale?

These four questions cover a significant part of the **IODASS** framework. The feasibility question is indeed crucial. Behind the feasibility question and by comparing with a *classic* iterative design process, we will identify if a notable supplementary work is needed to design and implement a piece of software using our method.

Software systems are meant to evolve over time and, as we have discussed many times in this dissertation, this is a top-level concern in software engineering. The combination of design rationale and formal model transformations has been designed with this particular issue in mind. We then had to evaluate if the

IODASS framework offers valuable assets to that purpose. As a side effect, the effectiveness of the languages as communication means could be partially verified, thought in a naive way inside a closed environment.

The expressiveness of the languages must be certainly evaluated. Even if we reused many concepts coming from the *state-of-the-art*, we introduce a list of new modeling elements. We have to ensure that no useless elements have been introduced into the languages.

We will now detail the protocol we followed for the comparative case study. We first present the framework under which the control group worked. We then discuss about the profiles of the participants. Afterwards, we detail the process we followed to avoid some *randomizations* effects that would have maybe lead to groups with consequent differences in modeling competencies. We introduce the Goal-Question-Metrics we defined for the case study. Last, we describe the case study itself with the content of the document we gave to the participants and the expected deliverables.

7.2.1 Select the Control Group

Throughout our research, we tested the **IODASS** framework and languages over toy examples to identify its lacks and improve them iteratively. At the moment we had a *stable release* of the languages, we decided to confront them to a recognized industrial standard. This choice was particularly decisive to avoid *comparing apples and oranges*. We needed to select a language with the same level of expressiveness to:

- model architectural elements that compose a software system, including the infrastructure/deployment
- list relevant requirements in a semi-formal manner, with relationships between them
- draw relations between requirements and architectural elements too
- add design rationale directly on model elements

As another main criterion, the question of the tool support was also crucial. We developed our tool suite within Eclipse, it looked primordial to identify a modeling language available in the Eclipse ecosystem too. Different environments could have lead to differences in the takeover of the modeling environment, even if, obviously the concrete Eclipse plugin under which the participants had to work was different, but, at least, parts of the tools were identical.

A last criterion was the absence of earlier knowledge in one or the other languages, again to avoid significant discrepancies in the participants' learning curves. This way, the study could not be biased by previous experience in one or the other language since they were no such preceding knowledge.

For all these reasons, we chose the OMG SysML [OMG, 2012d] standard since it answers positively to all our criteria. Its modeling facilities allow structural modeling of a software system together with its requirements. Design rationale may be added to model elements and relationships can be drawn between requirements and to model elements too. Also, many SysML tools exist as Eclipse plugins.

Last, SysML, with its ability to refine *Block Definition Diagram* (BDD) into *Internal Block Diagram* (IBD) where each `block` may be further defined into its own BDD and refined into IBD, particularly suited to be used in an iterative design process. Both the control group and the group under-study could use the same architectural design method, complying with Hofmesister's *general model* discussed in Section 6.3 from last chapter.

7.2.2 Participants' Profiles

Even if all participants were students from the University of Namur between 21 and 25 years old, their level of experience was not fully equivalent. First, some of them followed a so-called *"Professional"* Bachelor degree in a Higher Education school where the curriculum focuses priorly on professional competencies, *i.e.* programming and technical skills. These students, depending on the actual schools they came from, often had a larger experience in developing software, but missed some theoretical background, that they have to catch up during a *transition* year when they come at the university. During that special academic year they have to attend upfront the Master's degree, they follow a list of modeling and object orientation courses where they are taught the theoretical aspects on which we partially rely for this case study.

All students followed a course on theoretical aspects of project management and were taught on software development cycles, mainly recent Agile methods like Scrum [Beedle et al., 1999; Schwaber and Beedle, 2001]. They actively practiced the Scrum development method for a mid-scale team project that required requirement analysis, system design, planning, coding, testing and even *vendor* skills. For one of this project, the students worked full-time for two months to build a three-tier system, involving the development of a web-based front-end, writing business logic code with distributed communications and designing relational databases. A simulated evolution phase was also organized after a first delivery of their software, so they could have a first contact with maintenance-related problems. They also had other software development individual and team projects each academic year of their cursus.

Because we organized the case study inside a course of the Software Engineering option in the Master's curriculum, the participants were almost equally composed by students from the first or the second year. Second year's students had a 4 months internship in research centers, internally or externally, or even in the industry, as part of their curriculum.

7.2.3 Classification Phase

For the comparative case study, we had to divide the participants in two groups, one that will work with our framework, one with a *control* method and language. Also, to raise the amount of output data to analyze, we decided to create teams of two students, so that we could have six individual results for both groups. Because of the aforementioned variation in terms of experience and because the amount of partici-

pant was not statistically significant, we decided to organize a preliminary phase where we evaluated their modeling competencies in order to build groups of *comparable* modeling skills. With twenty-four students, the probability to have all second year students or all most skilled students in on or the other group was too large, such that the results of the case study could have been biased or uninterpretable.

Preliminary in-class case study

For this classification phase, we gathered all students in a large room during a 2-hour lecture. They received a document with the description of a simplified vehicle inspection system. They were asked to draw a UML class diagram from the requirement analysis present in the document. Were specified in that analysis[1]:

- an informal description of the system
- three UML use case diagrams with the expected functionalities
- the complete description of each use case scenario in a textual form
- clear instructions on the expected level of details for the class diagram

The general purpose of the system was to automate the vehicle inspection process with a mobile application for the inspectors, a local system for the inspection center, and a web-based interface to make appointments for car owners. As an example, the use case diagram of the mobile application is reproduced in Figure 7.1.

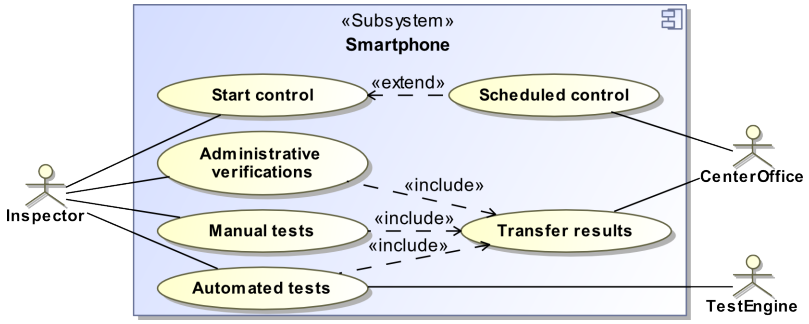


Figure 7.1: Use case diagram of the mobile application for the inspection system

An `Inspector` can *Start a control*, that can be in the list of *Scheduled control* (i.e. control for which car owners have made an appointment). He may perform the *Administrative verifications* and the *Manual* and *Automated tests*, all of them being *Transfered* to the `CenterOffice`. The *Automated tests* also involves the `TestEngine` as a secondary actor.

Similar use case diagrams were specified in the document for the functionalities of the `CenterOffice` and the `WebSite` subsystems. The `CenterOffice` allows the `Inspector` to *Finalize the inspection* and, if needed *Print a new notification*. Also, a `Scheduler` *Sends daily reports* to a `BackOffice`. A `CarOwner` may *Register* and

[1] The given document was written in French, as all students were native French speaker, in order to avoid a possible bias in differences in English comprehension.

Connect on the `WebSite` of the inspection system to *Make an appointment* for his car to be verified. He may also *Disconnect* or *Unregister* from the `WebSite`.

All of these use cases were detailed in a textual scenario. Table 7.2 gives an example for the *Finalization* of an inspection.

Name Finalize inspection
Summary The car owner finished the inspection and goes to the check-out desk.
Actor Inspector.
Precondition The system of the inspection center is working properly. The inspection linked to the car owner at the check-out desk is completed (every results being transmitted).
Postcondition The owner receives its inspection summary and paid the requested amount for the inspection.

Description

Inspector	CenterOffice
1. The inspector selects the plate number linked to the car owner at the check-out desk.	
	2. The center office system displays the amount to pay.
	3. The center office system prints the inspection summary.
4. The inspector confirms the payment made by the car owner.	
	5. The center office system saves the confirmation of the payment.
	6. The center office system displays the list of vehicles currently under inspection.

Alternatives 2.b The results associated to the inspection are not all received yet. The car owner is invited to wait until all results are effectively received by the center office system (back to scenario point 1.).

Figure 7.2: Textual description of the *Finalization* use case

At the end of the document, the students were required to draw a UML class diagram of a *first-draft* design. An example *Client-Server* class diagram was depicted to clearly state the expected level of details. They had to draw their own diagram individually on a A3 paper we gave to them. They also were spread all over the room to minimize cheating possibilities. We gathered all copies at the end of the lecture and made a ranking of their copies.

Judging and ranking phase

Inspired by a *judging* protocol by Jones [Jones, 1983], we asked three researchers to classify the class diagrams produced by the students. Two of them were internal researchers from the *PReCISE Research Center*, and one was an external senior

researcher from the *Computer Science Laboratory of Lille* (LIFL). First, they individually drew their own representation from the same document given to the students. Second they classify the students' copies in categories, still individually, based on their own criteria.

After this individual classification round, we gathered all three researchers around a table and discussed their classifications. It appears they all used similar categories, which were:

category 1 syntacticly and semantically incorrect
category 2 syntacticly correct, but semantically incorrect
category 3 syntacticly and semantically correct, but incomplete
category 4 syntacticly and semantically correct, and complete

For all copies, the three judges assigned a numerical value corresponding to 1 for diagrams belonging to the first category and 4 for the ones in the fourth category. We then calculated the *truncated arithmetic mean* value for all diagrams, denoted tam, and built three new categories based on these tam values:

low $tam < 2$ with 7 students
mid-range $2 \leq tam < 3$ with 11 students
high $tam \geq 3$ with 6 students

In order to equalize as much as possible the design competencies between the future groups and teams, and at the sight of the above categories, our idea was to define four *hats* of students to let a bit of freedom to them to make their pair teams for the remaining of the study. We then raised one students from the *low* to the *mid-range* category. Three students had the same (un-truncated) arithmetic mean (2.83 exactly), so we used their previous results in a software modeling course[2] to decide between them. The higher result was moved in the *mid-range* category that still needed to be split in two.

This category was divided based on their real arithmetic mean value obtained at the classification phase, then on the students' results in the software modeling course, last using their final results for their 3^{rd} bachelor. We ended up with four categories. We finally *pseudo-randomly*[3] created the four *hats* based on these categories, *i.e.* half of each category was assigned to each group, such that half of the first and second categories were grouped in one hat (called *I-a*), the remaining halves in a another one (*S-a*), one half of the two last categories in the third *hat* (*I-b*) and the remaining of students in the last one (*S-b*). Students from the *S-a*, resp. *I-a*, hats could finally choose a teammate in the *S-b*, resp. *I-b* one.

7.2.4 Preparing the Case Study

Upfront the beginning of the case study, both groups received an introduction to the languages they had to use. They received a 2-hours lecture where the main

[2] This course is part of the 3^{rd} bachelor academic year, called Analysis and Modeling of Information Systems where students learn about Object Oriented and UML modeling.

[3] We rolled two twenty-faces dices for each students, even results were assigned to *I* hats, odd ones to *S* hats. We particularly paid attention to have a fair distribution between the *I* and *S* groups, based on the aforementioned students' results and classification phase, that is the reason why we define the process as *"pseudo"* random.

principles of their assigned languages were presented. These lectures were organized separately, such that the participants had no practical knowledge over the other languages.

During the lectures, the participants were taught with the needed constructs to model a system, list requirements, define relations between them and add design rationale. The presentations followed the same template, were given by the same person, and were completed with a *screencast* where the installation of the plugins were presented, with the definition of a *helloworld* model to illustrate the main functionalities. The students also received a list of pointers to external references. After these introductory in-class sessions, the participants received a dedicated document with the case study, clear directions and timing details.

7.2.5 Case Study Description

The system the students were required to implement is the one we use in this dissertation: the book library, summarized in Figure 3.8. Instead of formalizing the functionalities as UML use cases, which have maybe biased the study since SysML reuse use case diagrams, we specified all requirements in a textual form, as depicted in Figure 7.3. Likewise the preliminary phase, this descriptive document was written in French, to avoid possible bias since all participants were native French speakers.

Ten requirements were listed in the document, five for the `OnlineLibrary` (prefixed by **OL**), four for the `Bookstore` (**BS**) and one for the `ParcelDelivery` (**PD**). All these requirements were very detailed in order to let the participants to get familiar with the modeling languages and the tools. A couple of simplification hypotheses and general advices were also stated in the document in order to keep the time limit of the implementation in the course's frame.

- one book is sold at a time;
- the amount of available exemplars of a book must not be considered;
- the effective book delivery is out of the scope;
- all systems are reliable, no crash recovery scenario must be considered;
- no database is needed, an XML initialization file, with a parser API and plain Java objects were offered to the students;
- no other functionality is required;
- no particular graphical design effort is required for the web pages.

After this first version, a simulated evolution phase was asked to the participants, with a new list of requirements, less detailed but still precise and unambiguous, listed in Figure 7.4.

For both phases, the participants were required to gather their models in a report where they detailed the iterative process they followed and documented their models. Together with the textual reports, the model sources (eclipse projects) were also required and the Java/JSP source code they wrote to implement the system. For the second phase only, the participants also wrote an evaluation report where they could criticize the languages they manipulated for the case study around the following general topics, each of them detailed with precise questions:

- expressiveness of the modeling constructs;

OLFunc_1	The system shall propose to the customer a catalog of books identified by an ISBN number, a title, one or more authors and a selling price.
OLFunc_2	The system shall record a book selling to a customer identified by its name, surname, street, number, box if existing, postal code, city and country.
OLFunc_3	When a book is sold to a customer, the system shall contact the bookstores to determine the lowest price for that book by organizing an auction between the bookstores. During the auction, every store sends its best price lower or equal to the current auction price. The auction ends when no more store can lower its price. If the auction finishes in a draw, the system shall choose randomly between the stores with the lowest price.
OLFunc_4	When a book is sold to a customer, the system shall contact the deliverer to ask the book delivery to the customer. The system sends the book ISBN number, the complete details of the store and the complete details of the customer to the deliverer.
OLNFunc_1	The system shall be available to customers on the internet.
BSFunc_1	The system shall send a catalog of books identified by an ISBN number, a title, one or more authors and a stated selling price.
BSFunc_2	The system shall permit to configure the minimal benefit margin on a book selling, expressed as a ratio of the cost price. This ratio is global to all books.
BSFunc_3	Depending the current auction price of a book, the system shall send to the library a new price, lower or equal to the current price, or a null value saying that the system is not able to propose a new price that comply with its minimal benefit margin.
BSFunc_4	The system shall record a book selling identified by its ISBN number for a given selling price.
PDFunc_1	The system shall record the delivery of a book identified by an ISBN number to a customer identified by its complete details.

Figure 7.3: List of requirements for the online book library (first phase)

OLNFunc_2	The system shall expose its catalog via a web service that will be used by mobile applications.
BSFunc_5	When an auction takes place, the system shall contact the other stores in order to determine the lowest selling price for a book sent by the library. The system offering the lowest price shall directly contact the library to make itself known.
PDFunc_2	When a book is delivered, the customer may withdraw his order. In case he withdraws, the system shall return the book to the store. When the book is returned to the store, the store shall contact the library to create a credit note equal to the book price for the customer.

Figure 7.4: List of requirements for the second phase

- added-value of modeling constructs as part of an Agile design method;
- easiness of model evolution and maintenance;
- documentation and its usefulness for model maintenance.

They were also asked to express in that evaluation report any suggestion or remark they judge useful or accurate.

7.2.6 Paper-Based Survey

To capture the *feelings* of the participants regarding the topics mentioned at the end of the previous section, we decided to crosscheck their free-format remarks with a structured paper-based survey. The survey was conducted in classroom and filled in individually by all participants. We reproduce the list of statements in Figure 7.5

(1) The languages constructs allow to represent:
  a) the expected functionalities of the system.
  b) the technological and communication constraints.
  c) the physical constraints related to the deployment.
  d) the non-functional requirements.
(2) The modeling language coupled to an agile development method as the one used during this laboratory offers an added value:
  a) to manage the complexity of the system-to-be.
  b) for the traceability of the requirements in terms of functionalities to implement.
  c) for the correctness and completeness of the implementation (code) of the system.
(3) The structural constructs impacted by a modification of a requirement can be identified quickly.
(4) The structural constructs impacted by a modification of a requirement can be identified at a glance.
(5) The language offers the necessary constructs and mechanisms to write an accurate documentation.
(6) The written documentation allows to efficiently comprehend the system within the framework of a modification of the system.
(7) During the second phase:
  a) a major work was necessary to re-understand the architectural concepts of the system.
  b) the modeling languages eased the structural changes linked to the new functionalities to implement.

Figure 7.5: List of questions of the paper survey

Inspired by the discussion made by Krosnick and Presser over the many methods present in the literature [Krosnick and Presser, 2010], each statement could be evaluated on an unmarked differential scale with only *fully disagree* and *fully agree* marks on each side. The participants could draw a line wherever they estimated it was appropriate. We found this method particularly suitable for our case because we wanted to compare two approaches. It lets a wide freedom to the participants since they may put their ratings on a continuous interval and it partially avoid re-ordering problems between questions when, for example, respondents want to show ordering relations between closed questions. Furthermore, fix point scales may lead to interpretation problems (what does a *"somewhat agree"* means, for example). An alternative solution would have been to add more graduations, but this solution

often leads to less spontaneous answers and increase the interpretation problems since the semantic variation between each point becomes even shorter [Krosnick and Presser, 2010].

7.2.7 Goal-Question-Metric Definitions

We now detail the results analysis framework we used for the case study within the Goal-Question-Metric method [Basili, 1992; Basili et al., 1994]. We decided to discuss the results of both the case study and survey outside a statistical framework because we did not have a statistically significant sample since the amount of software engineers is very large (which requires then a large sample) and we were conducting the study over students (which limits the generalization possibilities). We preferred to stick to objective metrics and discuss only over very large differences, instead of reasoning over statistically-sounded values with a statistically-insignificant sample.

Based on the four general questions we stated at the beginning of this chapter, we detail our evaluation objectives and deduct their metrics using the GQM approach where every identified *Goal* is decomposed into a *Purpose*, a quality *Issue*, a process *Object* and a *Viewpoint*. All goals are also completed with their addressed research questions between brackets.

As a first goal, we want to evaluate the feasibility of the overall approach. We are interested to verify whether it is effective to iteratively enrich a software architecture through formal model transformations.

Goal 1	*Evaluate the feasibility of a transformational architecture design method to design a software system* **[RQ 3.1]**
PIOV	Evaluate / the feasibility of iteratively transform a / software architecture model / from the project manager's viewpoint
Question 1	Is it effective to implement a software based on an architecture model created from stepwise formal model transformations?
Metric 1	Number of top-level functionalities correctly implemented

Second, we want to evaluate the quality of the DAD and ASR models regarding two criteria: the number of requirements for which we cannot find an architectural element that implements them and the number of decisions that are effectively documented by either a formal relation to other requirements/architectural objects or by some rationale.

Goal 2	*Evaluate the quality of architecture and requirement models using DAD-ASR languages* **[RQ 1.1,1.2,2.2]**
PIOV	Evaluate / the functional completeness of a / software architecture model regarding the expected model elements / from the architect's viewpoint
Question 2	Does the produced architecture models contain all expected components and interfaces to fulfill the software's requirements?
Metric 2	Number of requirements without any responsible element.

Question 3 Are the newly introduced subrequirements correctly documented with their rationale?
Metric 3 Number of decisions regarding subrequirements with a meaningful explanation of their purposes (rationale).

Third, we want to investigate the recordings of the design process itself to understand its history.

Goal 3 *Evaluate the traceability of a transformational method regarding the history of the development process (planning-evaluation)* **[RQ 2.1,2.2]**
PIOV Evaluate / the actual implementation order of the / architecturally significant requirements / from the architect's viewpoints
Question 4 Does all development iterations have been *backlogged* for evaluation and traceability purposes?
Metric 4 Number of iterations reported with corresponding implementation plans.

And fourth, we want to evaluate the feasibility of our approach in maintenance or evolution activities.

Goal 4 *Evaluate the feasibility of a transformational method in maintenance and evolution activities of a software system* **[RQ 3.1]**
PIOV Evaluate / the feasibility of iteratively transform an / existing software architecture model / from the architect's viewpoints
Question 5 Is it effective to incorporate new functionalities in a software based on an architecture model modified by stepwise formal model transformations?
Metric 5 Number of impacted functionalities correctly implemented

7.3 Results and Discussion

We now details the result we gathered from both iteration phases. We first concentrate on the *metrics* we defined. Then we discuss the questionnaire results and we summarize the various comments received from the participants. Afterwards, we outline our findings regarding our four *goals*.

7.3.1 Results of the First Phase

The study was composed by two distinct phases, as detailed in Section 7.2.5. At the end of the first phase, we tested the functional correctness of the teams' prototypes and we calculated the aforementioned metrics, which are summarized in Table 7.1.

The first column gives the identification number of all teams, from *S1* to *S6* for SysML, and from *I1* to *I6* for **IODASS** . The second column shows the results of the *happy scenario* test. We deployed the prototypes according to a *readme* file the teams joined to their source code and models. This happy scenario regrouped the three

Table 7.1: Evaluation of the deliverables of phase 1

Team	Happy scenario	Impl.(M1)	Req.	Untr.(M2)	Decis.	Rat.(M3)	Iter.(M4)
S1	Stopped during auction	2	19	1	17	10	4
S2	No possibility to order	0	15	2	11	2	0
S3	Fully functional	4	10	0	10	1	0
S4	Fully functional	4	10	0	11	8	3
S5	Stopped after auction	3	11	1	16	4	0
S6	No possibility to order	0	10	0	12	2	3
$Q2_S$	n/a	2.5	10.5	0.5	11.5	3	1.5
I1	Fully functional	4	23	1	15	15	1
I2	No book delivery	3	30	1	28	28	5
I3	Fully functional	4	10	10	0	0	0
I4	Stopped after auction	3	14	2	9	8	8
I5	Fully functional	4	41	3	29	29	4
I6	Compilation failure	0	27	3	24	21	5
$Q2_I$	n/a	3.5	25	2.5	19.5	18	4.5

top-level functionalities divided in four sequential steps: the ordering of a book, the start of an auction, the end of an auction that lead to the cheapest price and the notification of the book delivery. The maximum amount of functional steps correctly implemented by the prototype is shown in the third column (*Metric 1*).

The next columns concern the quality of the models and the decision traceability. The number of requirements listed in teams' models is given in the fourth column (*Req*) and the untraced requirements (*Metric 2*) is shown in the fifth column. Those values were identified from the SysML requirement tables and **ASR** models. In both types of models, we were looking for any relation linking requirements to each others or to model constructs. However, since all requirements in **ASR** models must be *assigned to* a model element[4], we excluded those relations from the *Metric 2* and only counted the other types of relations.

The amount of described decisions from models and reports is reproduced in the sixth column, together with the amount of identified rationale (*Metric 3*). We carefully analyzed the models and design reports to extract all design decisions and their rationale. In some cases, they were clearly identifiable, but in most cases for *S-Teams*, those data had been manually extracted from free-text explanations. For **ASR** models, we counted the *meaningful* rationale only, *i.e.* the rationale that gave some substantial information on the reason why such a decision was taken.

Last, the number of design iterations, as recorded or identified in the teams' deliverables, is shown in the last column (*Metric 4*). As for the previous metric, these values were mostly manually extracted from careful analysis of the textual reports.

The median values (*Q2*) of each column are also calculated to compare results between both groups. We use median values instead of arithmetic means since the sampling is rather small and we want to concentrate on *central tendencies*, instead of purely mathematic values.

On the functional side, two *S-Teams* delivered a fully working prototype and one more *I-Team* did so. There was no possibility to start the book ordering for two *S-Teams* and one *I-Team*. The median implemented steps in our happy scenario is

[4]This is a mandatory feature, as explained in Section 4.3.2, so we investigated other types of relations, otherwise, by default all requirements would have been traced to a model construct in our counting.

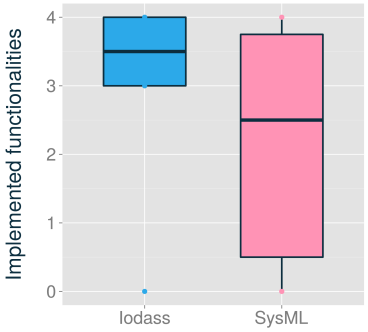


Figure 7.6: Correctly implemented functionalities

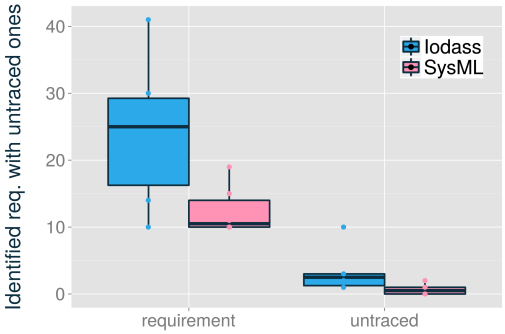


Figure 7.7: Identified requirements with untraced links to model elements

slightly higher for *I-Teams* where *3.5* steps were correctly executed against *2.5* for the *S-Teams*.

The amount of requirements listed by the *S-Teams* and *I-Teams* are more contrasted. Three *S-Teams* and one *I-Team* only listed the ten requirements present in the description document they received. A fourth *S-Team* added one more sub-requirement and the remaining two identified respectively *5* and *9* more requirements. Except the one team that did not refine any requirement, *I-Teams* created from *4* to *31* sub-requirements or alternatives. As a median, *10.5* requirements were identified by *S-Teams* and *25* for I-Teams. Regarding the untraced requirements, our *Metric 2*, three *S-Teams* had no such requirement and the other three showed a maximum of *2*. On the other side, the median amount of untraced requirements is higher for the *I-Teams* with *2.5* against *0.5* for *S-Teams*, even if for both groups, the value is rather low.

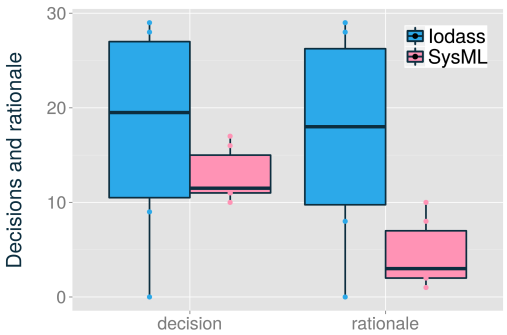


Figure 7.8: Number of decisions and rationale

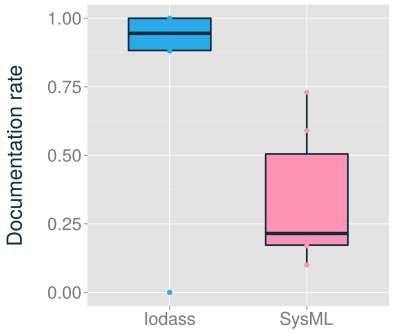


Figure 7.9: Documentation rate

Regarding the decisions and their rationale, we observed a median of *11.5* decisions for *S-Teams*, going from *10* to *17*, for a median of *3* rationale (from *1* to

10), which corresponds to a *median documentation rate*[5] of *0.215*. The *I-Teams* recorded from *0* to *29* decisions for an median amount of *19.50*. Almost all these decisions were documented by as many relevant justifications, *0.945* as median documentation rate.

Last, three *S-Teams* gave some details over the iteration design cycle they actually followed, for four *I-Teams*, one of them recorded only one iteration which does not really correspond to a so-called iterative process. The *S-Teams* needed *1.5* iterations, but since only half of the teams gave some details over the iterative process, this value does not really make sense and the typical amount of iterations may be considered between *3* and *4*, as the other *S-Teams* recorded. Although, the median value for the *I-Teams* is a slightly higher to that range, with *4.5* iterations.

7.3.2 Results of the Final Deliverables

For the final release, we tested the teams' prototypes in a different manner. Each team had a 10 minutes slot to demonstrate the three new functionalities about the web service interface, the modification of the auction mechanism and the issue of the credit note in case of a withdrawal. We were also interested in investigating the level of details the teams provided for the confirmation page as a feedback to the customers. No clear demand was formulated in that sense, but we want to evaluate if, as a side effect, our structured way of recording and decomposing requirements has an impact on the functional completeness. The modification of the auction mechanism induced a significant rework at the architectural level because the responsibility needed to be transfered from the library to the stores, with the winner calling back the library by itself. By contrast, the withdrawal and credit note were an isolated evolution of the system, since they were a new functionality that was added at the end of the happy scenario depicted in last section.

Since all teams correctly implemented the web service, we do not show it in the final results reproduced in Table 7.2.

Table 7.2: Evaluation of the final prototypes and deliverables

Team	Feedback	Auction	Credit	Impl.(M5)	Req.	Untr.(M2)	Decis.	Rat.(M3)	Iter.(M4)
S1	Medium	Fully	Incomplete	5	23	2	19	12	0
S2	None	Partially	Complete	3	17	1	13	2	0
S3	Medium	Fully	Incomplete	5	15	0	13	4	0
S4	Basic	Fully	Complete	5	14	0	24	7	0
S5	Basic	Fully	None	3	15	1	18	5	0
S6	Basic	Partially	Complete	4	18	1	16	4	2
$Q2_S$	n/a	n/a	n/a	4.5	16	1	17	4.5	0
I1	Medium	Fully	Complete	6	27	3	24	24	1
I2	Basic	Fully	Complete	5	35	1	31	31	3
I3	Medium	Fully	Complete	6	17	14	3	3	1
I4	Basic	Fully	None	3	17	3	11	11	2
I5	Basic	Fully	Complete	5	56	5	51	48	3
I6	Complete	Fully	Complete	7	38	4	34	28	7
$Q2_I$	n/a	n/a	n/a	5.5	31	3.5	27.5	26	2.5

[5]The documentation rate is calculated as $\frac{\#Rationale}{\#Decisions}$ to express the proportion of documented decisions.

Almost all columns are identical to Table 7.1, except that we introduced dedicated columns for all three aforesaid functional criteria: the user feedback, the auction mechanism and the credit note for a withdrawal. We also only considered the amount of correctly implemented requirements related to the evolution phase, as defined in our *Metric 5*.

To calculate the *Metric 5* value, we rate the feedback as follow:

None=0 no feedback at all
Basic=1 very few details, partial customer, book or price details
Medium=2 almost all details shown, but still missing one or the other
Complete=3 all details about the customer, the book and the price were given

The auction is rated in a similar way:

Partially=1 responsibility correctly transfered, but in a synchronous way
Fully=2 asynchronous auction with the callback

And the credit note:

None=0 no credit note issued
Incomplete=1 missing details in the note
Complete=1 all details with price and customer data

We reuse the *Metric 4* for the last column concerning the number of iterations, but only focus on the second phase, since we could isolate it from the reports and models when they were specified.

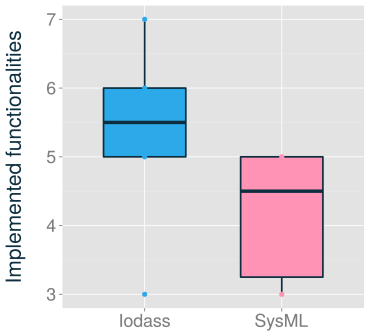


Figure 7.10: Correctly implemented functionalities

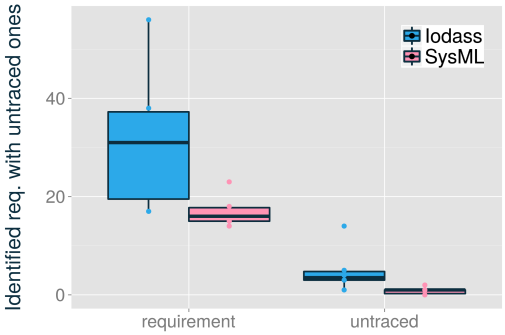


Figure 7.11: Identified requirements with untraced links to model elements

The median ratings for the functional correctness are quite close for both groups with *4.5* for *S-Teams* and *5.5* for *I-Teams*. But we can note that the *6 I-Teams* implemented correctly the new auction mechanism against *4 S-Teams* and *5 I-Teams* provided a complete credit note against *3 S-Teams*.

The *Metric 2* follows the same tendency as we identified at the end of the first phase. Significantly more requirements have been listed by the *I-Teams* with a median of *31* requirements, confirming a more systematic decomposition and alternative exploration under our framework. But as for the first phase, more untraced requirements were also recorded, with a median of *3.5*, compared to the only *1* for SysML teams.

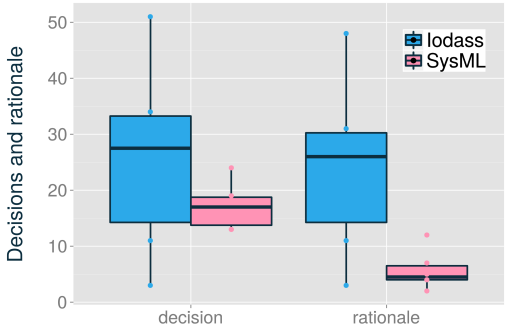


Figure 7.12: Number of decisions and rationale

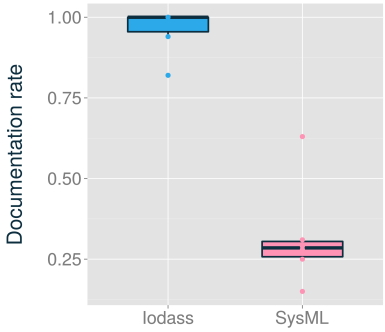


Figure 7.13: Documentation rate

The amount of design decisions and rationale also increased in both groups, *S-Teams* recording from *13* to *24* decisions (mean of *16*) and I-Teams from *3* to *51* $Q2_I$=*31* decisions. The median documentation rates of decisions increased to *0.285* for *S-Teams* (median of *4.5* rationale) and to *1* for I-Teams, with $Q2_I$=*26* rationale.

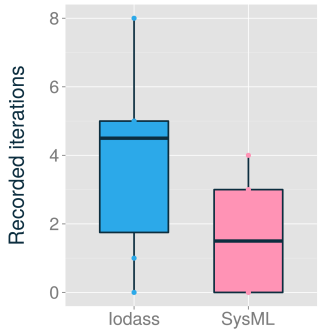


Figure 7.14: Number of iterations for the first phase

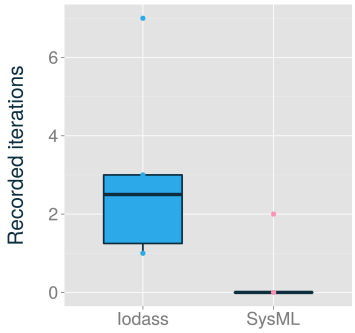


Figure 7.15: Number of iterations for the final phase

Last, *1 S-Team* explicitly specified the iterative steps it followed for all *6 I-Teams*, but with *2* of them recording only one iteration. We could not identify any explicit or implicit description of the design steps for the remaining *5 S-Teams*. The median amount of iteration step for *I-Teams* is *2.5*, from *1* to *7* iterations.

7.3.3 Participants' Remarks

In their evaluation reports, all participating teams formulated some remarks over the languages expressiveness, their added-value as part of an Agile-oriented method, the easiness of model evolutions and the usefulness of documentation.

SysML

All *S-Teams* highlighted the large flexibility of the SysML `block` construct, they often characterized as excessive and even confusing. They particularly had difficulties to decide when to stop the refinement of *Block Definition Diagrams* into concrete *Internal Block Definitions*. They were also puzzled by the semantic freedom of the `block`, even if they also noted it was an asset in some cases.

They also complained about the limited design decisions and rationale traceability. This remark is partially related to a bug of the Obeo SysML plugin[6] we used. Prior to the case study, we evaluated a list of available Eclipse plugins for SysML model editing and the tool offered by Obeo, thought a commercial one, was working correctly on our testing environment. However, as the case study started, they released a new revision, making the prior one not available anymore, and this new version contained a visualization bug. The *requirement cross-table*, where requirements are summarized with their *satisfying* `blocks`, was partially not readable, the names of the `blocks` being hidden in the table headers. All *S-Teams* were not affected by this bug, or not exactly in the same way, regardless the Eclipse build or operating system they were using. However, the possibility to add `rationale` annotations to SysML model elements was perfectly working, but none of the teams actually used it.

The last collegiate remark concerns the communication definition facilities between `blocks`. The majority of *S-Teams* explained that they did not fell comfortable with the communication semantics of SysML. They had difficulties to decide whether they needed `flows` or `interfaces`. When we took a look at their models, only one team correctly defined the communication links between `blocks`, the other teams either misused some constructs or even did not provide any detail on the communication.

IODASS

The prototype used by the participants was not preserving the comments present in the **DAD** models after a transformation and all *I-Teams* were disappointed by this disappearance. All participants considered in-code comments as important pieces of documentation, but the *serializer* in charge to print the transformed models into text files did not preserve code comments. The last version of the **IODASS** transformation engine now directly uses the *serializer* generated by Xtext from the language grammar, which, among other benefits, conserves the comments in the model.

The second main remark concerned the textual notations of **IODASS** models. Many of them had some troubles with the textual notation during the first phase and draw graphical representations aside. All participants were used to graphical notations, *e.g.* UML models, and the **IODASS** textual syntax was unsettling to them, even if they got used to it as the case study was going. Many of them advocated for a combined textual-graphical notations where the big picture may be browsed graphically and the details textually.

[6] http://marketplace.obeonetwork.com

Last, part of the participant noticed that they had little usage of the **Deployment** constructs. They explained that the amount of deployment constraints were not sufficient to judge of their utility or not. They advocated for a reusable library of properties to further refine the constructs semantics, which was lacking at the time of the study and have been added in further releases of the **DAD** language.

7.3.4 Questionnaire Results

The last source of feedbacks and evaluation criteria can be found in the results of the paper survey we conducted at the end of the case study. The unmarked scale was going from *fully disagree=0* to *fully agree=5*. All statements but the *7(a)* were written in a *positive way*, so higher values are better. The 7(a) stated that *"a major work was necessary to re-understand the architectural concepts of the system"*, so higher values are worst.

Figure 7.16 gathers the results we calculated from the students questionnaires in a bar chart graduated from *0* to *4.5*.

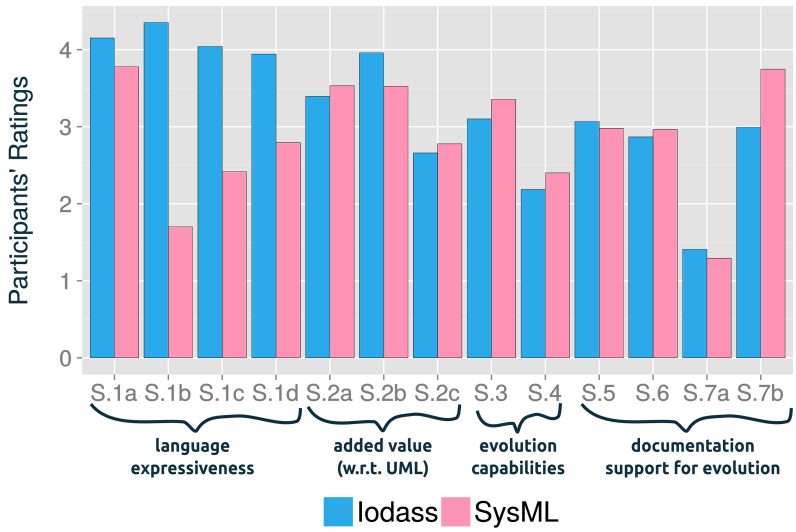


Figure 7.16: Results of the questionnaire-based survey

From this chart, we can identify three main tendencies: the language expressiveness is higher for **IODASS** languages than SysML, especially regarding the technological, communication (*statement 1b*) and deployment (*statement 1c*) constraints. A slightly less significant, but still visible, improvement of the **IODASS** expressiveness concerns the non-functional requirements (*statement 1d*).

The second tendency concerns the evolution capabilities (*statements 3 to 7b*) where, even if we rely on textual models that were perceived as less intuitive by the participants, our framework performs almost as well as SysML. However, a somewhat

noticeable difference is present for the last question related to the easiness induced by the modeling languages for structural changes.

As a last tendency, except for *statement 4* related to the relative easiness to identify changes in a model "at a glance", and *statement 7a* that was formulated in a negative manner, all participants' ratings for the **IODASS** languages are above *2.5*. The *statement 7a* has been rated below *1.5*, which is also an encouraging result, since the participants estimated that no *major work* was needed to come back into an existing model.

Finally, *statements 2* were investigating the added value regarding the ability to manage the complexity (*2a*), the traceability features (*2b*) and the correctness of the produced code (*2c*). Except for the traceability feature where the **IODASS** ' rating is a bit higher, the values expressed by both groups are very close, with a very slight higher ratings in favor of SysML as a modeling mean in Agile development. However, those three differences are insignificant, but we may notice that all ratings are above *2.5*, which indicate a rather satisfactory rating, except maybe about the link to the code. Concerning this *round-trip* feature, those low ratings are most probably related to the absence of code generator for both modeling tools, as expressed by a large majority of the participants in their final evaluation reports.

7.3.5 Discussion

Our first goal was the evaluation of the feasibility of our approach to design a software system. From the functional tests we executed at the end of the first phase, the results were slightly better under our framework with *5 I-Teams* that delivered a (partially) functional prototype for *4 S-Teams*. One more *I-Team* delivered a fully functional one. If we also consider the results of the second phase too, the same tendency can be observed with marginally better results for the **IODASS** teams. The ratings in *statements 2* from the survey also partially confirm this aspect, since the participants found the modeling languages appropriate to manage the complexity of a system and trace requirements to model artifacts. We may then conclude that

> *it is feasible for master students in software engineering to successfully implement a system that have been modeled with the* **IODASS** *framework under our controlled environment.*

Second, we were interested in checking if a model could fulfill the list of requirements with sufficient justifications for the decisions taken to build it. We looked at the amount of sub-requirements produced and how much of them could not been traced back into the architecture model. We identified that the *I-Teams* produced more sub-requirements than the *S-Teams*. However, the amount of requirements without *meaningful* extra traces is higher for the *I-Teams*. Those missing links were mainly related to non-functional requirements that students had troubles to either implement as a transformation or relate to an existing `Interface` with either `Implementation` or `Usage` decision. After some analysis of the involved requirements and if we do not take into account the team that did not actively followed

the **IODASS** iteration process, we observed that those untraced requirements were related to implementation details (like the OLNFunc_2 regarding the replication of an existing interface as a web service) that the students actually implemented in their Java code but did not produce dedicated model constructs for that purpose.

But, as said earlier, an `Asr` is always assigned to a **DAD** architectural element (equivalent to the SysML *satisfy* link), but those links were not counted since they are mandatory and we wanted to estimate how many requirements finally remained unbound to other types of more concrete decisions. The same tendency appeared in the second phase too, with a higher number of requirements produced, but also a higher proportion of them remaining untraced. Furthermore, some types of relations were not implemented yet in the prototypes used by the students. The implication, exclusion and impact were not present in the **ASR** language yet and were added later on, based on the literature and the many advices and reviews we received after the study. For all these reasons, we can only observe that

> *there seems to be a positive influence of our systematic recording and decomposition of requirements, but more requirements stay unrelated to architecture model constructs.*

The other part of this second goal concentrated on the amount of decisions taken to design a software system and the amount of justifications that sustain a model. On the one hand, these metrics were easily counted from **ASR** models because dedicated constructs exist inside the model itself. We still discarded the *"dummy"* rationale that were empty, had no sense or gave no real justifications of the decisions. On the other hand, some types of decisions had not been recorded directly inside SysML models (no usage of the `rationale` annotation) and requirement tables, so they had to be manually extracted from textual descriptions and justifications given in the participants' reports. From this analyze, we recorded a significant amount of decisions like *"the system is divided in three components."* or *"The communication between the components is done over CORBA."* without justifications. As advocated many times, we do believe informal justifications are not suitable to effectively record design rationale. But, from the results of the survey for *statements 5, 6* and *7(a)*, and from the evaluation remarks expressed by the participants, we noticed that more can still be done to increase model comprehension and documentation. We already updated our tool suite to keep the comments present in the code as they were considered as important pieces of documentation. We may then conclude that with the significant improvement in the documentation rates we observed in the case study,

> *the systematic recording of design decisions increases the amount of design rationale, but its positive impact on model comprehension is unsure.*

Our third goal focused on the history of the design process and the traceability of system design phases. We observed significant differences between both phases where the *S-Teams* either implemented all changes in the model in one iteration

or neglected to provide this piece of information. On the other side, *I-Teams* massively followed the iterative process by creating distinct model versions. Since for this fourth question, we are concerned by the traceability of the history under our framework only, we may conclude that

> *in most cases, the participants follow an iterative design process and the history of the successive changes are completely recorded.*

The last goal is closely related to the first one where we wanted to evaluate the feasibility of our framework, but here, for evolution and maintenance activities. As for the functional correctness we noticed at the end of the first phase, the final prototypes were slightly more accomplished. At the opposite of the first systems, all final software were at least partially implemented, with one group on each side having bypassed the generation of the credit note. Even if the test results for the modified functionality, i.e. the auction process that was transfered from the library to the stores, were better for *I-Teams*, the needed modifications were perceived as a bit less intuitive by the **IODASS** teams, as shown by *statements 4*, *6* and *7(b)*. At the sight of the participants' remarks, we may believe the textual syntax played a role in this lower ratings in the survey. However, since *5 I-Teams* delivered functional prototypes, with at least partial implementations of all requested requirements, we may conclude for this fifth question that

> *it is feasible for master students in software engineering to successfully modify an existing functionality with noticeable impact at the architectural level and add new functionalities to an existing system specified and documented within the* **IODASS** *framework under our controlled environment.*

We also wanted to gather the participants' opinions regarding some aspects of our languages and especially their expressiveness. In the survey we conducted, *statements 1* were devoted to evaluate that aspect and the results are fairly satisfying, since the gain comparing to SysML is noticeable, especially for the communication and deployment facilities. So either if the participants expressed some reserves regarding the textual syntax at first, they felt more comfortable with it and even rated it as accurate to represent architectural constructs and their requirements.

7.4 Threats to Validity

In the present chapter, we reported about the comparative case study we conducted over master students in software engineering. We also drew a set of conclusions, based on the observations we made, the functional tests we run over their prototypes, and the results of the survey we organized. We will now evaluate the validity of these conclusions by following the classification of threats to validity, as identified by Cook and Campbell [Cook and Campbell, 1979] and extended by Wohlin *et al.* [Wohlin et al., 2012]. Some of the threats are not detailed in the following section, those are the ones which were not relevant for our case study, like the *Mortality* for example.

7.4.1 Conclusion Validity

The question of the conclusion validity deals with the ability to draw valid conclusions from the outcomes' observations during an experiment [Wohlin et al., 2012].

Statistical-related threats

A couple of threats to validity relies on the statistical tests applied to interpret the outcomes of an experiment. Since the beginning of this study, we knew we could not build a statistically significant sample and that the profile of the subjects was not totally random. Therefore, we did not use any statistical framework, but stayed focused on objective metrics that could be interpreted *per se*. We have also been particularly careful in the conclusions we draw in Section 7.3.5.

Reliability of measures

For the count of design decisions and their rationale (*Metric 3*), we cannot ensure we did not miss one or another decision during the manual analysis of the *S-Teams*' reports. However, we proofread many times those documents and the differences between both approaches is so large that we may reasonably think that even if we missed a few decisions or rationale, the conclusions over the systematic recording of design decisions still stand.

The missing information regarding the *S-Teams* iterative process for the second phase had no influence on our conclusion. We were only interested to identify if the *I-Teams* effectively followed the iterative design process. These measures are fully reliable since they are directly available in the **ASR** models inside the delivered Eclipse projects.

Random irrelevancies in experimental setting

The main part of the case study was conducted outside classroom. We then cannot guarantee the participants did not communicate from teams to teams. However, we do not think such a communication may have influenced the results themselves. We obviously paid attention to the produced models and code to identify clones or cheating, which we did not found any proof of.

Random heterogeneity of subjects

The experiment was almost conducted *in a vacuum*. The participants profiles were rather similar, even if their concrete experiences were slightly different for some of them. Partially because we had a limited amount of participant and to try to tackle a possible randomization problem that would have lead to unbalanced groups, we organized the preliminary round to classify the students based on their structural design competencies. We believe this initial phase helped to this purpose, but as the amount of participants is limited and the initial phase was also limited, we may not ensure a completely homogeneous distribution of participants between teams and groups.

7.4.2 Internal Validity

The internal validity deals with the influence of the variables under study, such that an effect observed during an empirical study is not actually caused by an external factor out of control [Wohlin et al., 2012].

History

Both groups had no previous knowledge or experience with one or the other languages and had comparable experience with Agile-oriented design methods. So we may reasonably think the history had no significant influence on the case study.

Maturation

We believe that the subjects gained in experience during the experiment, since the study was partially organized as a pedagogical mean too. But this maturation in knowledge should not have influenced the results. In the other way around, some participants may have gotten bored or tired, even if we have heard no complaint from them during or after the study.

Instrumentation

In both groups, the tools used by the participants may have influenced the study, even if they were part of the same platform. For *S-Teams*, the Obeo plugin revealed to be unstable for some visualization purposes after an update and may have lead to a partial discouraging or tiring effect. For *I-Teams*, the prototype was still in early development stage, thought in a stable release, and was maybe not as user-friendly and eye-candy as commercial tools. The differences in modeling approaches, textual *vs* graphical, has also been considered when evaluating the results of the survey (regarding the maintenance-related statements) and we drew our conclusions accordingly.

Selection

The selection process was clearly done *by convenience*. As discussed in Section 7.1.2, students are an affordable source of participants to conduct studies, at the moment some precautions are taken and the study has a pedagogical goal too [Carver et al., 2010]. This convenient sampling was partially balanced by the initial evaluation phase.

Interactions with selection

For both approaches, we created multiple teams, such that possible differences in learning curves are balanced by the number of teams in each group. We clearly observed discrepancies between the groups of each approach, so we reasoned on median values, instead of individual results, to concentrate on *middle-range tendencies*.

Social threats

We especially paid attention to act with both groups the same way, without giving more attention to one or the other team. The presentations of the languages and tool support were organized separately, but exactly the same way with the same level of details by the same person. We cannot ensure that the teams did not discuss with each other, but we do not believe such interactions have influenced the study. If such an influence would have appear, the *S-Teams* would have probably paid more attention to document and refine their requirements in a more systematic manner than they actually did.

Also, as the participants were aware that **IODASS** was our framework since we had to point them to technical reports with details over the modeling languages, some students may have been too respectful with our languages, or at the opposite, they wanted to blast our framework by *over-rating* SysML which is an industrial standard. The exact repercussion of this *good-looking* effect is honestly complicated to determine. We can only observe that one team actually did not followed our framework at all, but no other conclusion may be drawn.

7.4.3 Construct Validity

Construct validity focuses on whether the experiment is adequately designed to measure what it is meant to evaluate [Wohlin et al., 2012]. They are either related to the design of the study itself or to social elements.

Design threats

We took some time to construct the case study and identify what we wanted to evaluate. In our case, the metrics were fairly obvious since we wanted to evaluate the feasibility of the approach through functional correctness of produced software and the possible gains in documentation. Both groups received the same instructions prior to the case study that we can summarize as *"develop a documented piece of software"*. But they were not aware of the metrics themselves. However, we did not use a very large amount of metrics, but we crosschecked our findings with the results of the survey and the evaluation reports written by the participants.

Social threats

We gave clear instructions regarding the expected deliverables for the study, such that even if they did not know the exact nature of the investigated metrics, they had a clear view of what we expected from them. Also, as the experiment was also part of their curriculum, they were required to participate seriously to the study, that was also evaluated at some points by external researchers or professor.

As clearly stated with the students at the beginning of the study, two aspects of their deliverables were evaluated as part of their grading in the embedding course: the semantics correctness of their models and the functional correctness of their prototypes. No other *quality*-related deliverable was taken into account in their final results.

7.4.4 External Validity

Last, the external validity is the ability to generalize the observed results to other communities, places or time [Wohlin et al., 2012].

Interaction of selection

We selected students in software engineering with a few experience in software development as study subjects. They all had a previous experience with a simulated realistic project. From Chapter 1, we already identified that the documentation of an architecture and a software system in general is crucial, and often lacking in industrial projects. So there is no indication that software engineers would have documented their models with a significantly higher proportion than the *S-Teams* under the same conditions. The design reports delivered by the students were already detailed, recording many decisions, but lacking much of the design rationale. On the other hand, we may reasonably think that junior analyst developers may have acted the same way as the *I-Teams* since their profile is relatively close.

Interaction of setting

We used *industrial-level* tools for our study, both being plugins of the Eclipse ecosystem which is popular in the software development community[7]. However, the Obeo commercial plugin was not free of bugs, even if we tested carefully prior the case study. Our tool was an early stable prototype, but was part of the study by itself. Except the aforesaid reserves concerning the tools, this threats should not have influenced the results.

Even if the size of the case study was small, the produced software was larger than a simple toy example because it involved some knowledge on design patterns (*asynchronous callback*, *observer* or *model-view-controller*) to be implemented in an effective way and the participants were required to demonstrate it was truly working. Therefore, we believe the case under study was large enough to already observe valuable results.

7.5 Wrap-Up and Conclusions over the Empirical Evaluation

In this chapter, we were interested in validating our transformation-centric approach and languages in contrast to an industrial modeling language used within a comparable design cycle. To this purpose, we set up a comparative case study where master students in software engineering had to design and implement an online library system. After a preliminary evaluation phase of the participants, we made twice six teams of two students of comparable structural modeling competencies and those teams had to implement the library system in two phases, the second one simulating an evolution of the first release of the system. After each phase, we

[7] See the Eclipse download statistics on http://www.eclipse.org. For example, on October 2014, the only standard build of the latest version released in September 2014 has been downloaded more than 385.000 times.

evaluated the functional completeness and correctness of the students' prototypes to identify if they had been able to deliver a functioning software based on models written within our approach. We also evaluated the documentation produced by all teams and especially the differences in the production of detailed requirements from higher-level ones and the justifications of their design decisions. In order to support our observations, we also analyzed the evaluation reports written by the participants as well as the results of a survey we conducted at the end of the study.

We identified that the functional completeness and correctness of the systems implemented by the teams using our framework were as good as the ones of the control group, even a bit better. We also observed a higher proclivity to refine higher order requirements into more detailed ones and a higher number of justifications as design rationale. However, we have to consider that we were experimenting in a closed environment with too few subjects to claim we have statistically-grounded results. We also identified other threats to validity which we tried to avoid with a systematic approach to define the protocol of the study, or which we had to recognize as possible biases to our results to temper our conclusions.

CONCLUSIONS AND PERSPECTIVES

Wrap-up of Contributions

Software architecture is a key aspect of the software engineering field as it offers a big-picture representation of a system. At this level, many constraints and requirements must already be taken into account, but staying independent enough to a particular technology to remain as stable as possible across evolutions. However, with subsequent changes that inherently arise over time, discrepancies may appear between a system and its architectural representation. Furthermore, the documentation of an architecture is primordial to keep control over it. Its rationale and the decisions that lead to a particular model are valuable pieces of information to understand its design on the long run. Thought, at some point, the technology and the deployment infrastructure influence a specific design and must be considered to avoid rework in case of conflicts between an abstract design and its target running instances.

At the beginning of this dissertation, after a review of existing component-based modeling languages, architectural knowledge recording techniques and transformation approaches, we identified a set of research questions.

RQ1.1 How can we represent SA models at different levels of abstraction ?

We proposed a domain specific language to represent software architectures from coarse-grained to fine-grained components. This **Definition Assemblage Deployment** language has been specified based on the many commonalities of existing component-based approaches, but with a couple of novelties that does make it *not yet another language*. Our proposal remains on a clear separation of the abstract **definition** of types of architectural modeling elements, **assemblage** of running specifications and **deployment** mappings to a target infrastructure.

RQ1.2 How can we represent SA models with flexible communication facilities ?

We defined a flexible **duck-typing-based** composition mechanism for software components. This mechanism relies on the separation between the specification of the communication protocol, the type of linkage between components and the concrete medium that interconnects targeted computation nodes. This binding enables to substitute *compatible* interfaces in an extensible manner by verifying the services signatures instead of relying on names only. Then, interfaces may be superseded based on partial semantics, instead of rigid inheritance mechanism. A taxonomy of

link types has also been defined to cover a wide range of connection possibilities between components, from point-to-point to broadcast-like, or even with user-defined load balancing strategies for the delegation of interface responsibilities.

RQ1.3 How can we represent SA models with deployment constraints ?

Our proposed language lets an important place to deployment constraints in order to specify target infrastructure in terms of nodes, gates and communication media. A dedicated mechanism for **user-defined property** has also been introduced to refine the semantics of all model elements in general, and especially deployment ones. Those properties open the door for model verifications in order to verify if an envisioned architecture design may be able to be deployed on an existing infrastructure or, at the opposite, identify computation and communication needs.

RQ2.1 How can we explicitly retain the link between SA models and related significant requirements ?

An architecture model is always bound to its requirements, **each requirement being assigned to an architectural element**. We specified a dedicated modeling facility to list architecturally-significant requirements in a **semi-formal manner** and attach them to any structural constructs, *i.e.* not only components, but also link types, interfaces or protocols, for example. In a sense, any element is then explicitly assigned to requirements, such that engineers always know who is in charge of the fulfillment of a (non-functional) requirement.

RQ2.2 How can we document the decision-making and argumentation processes when designing SA models ?

As engineers refine requirements or evaluate alternatives, they are encouraged to fill in the requirement listing with their decision-making process. **A set of relationships**, identified from the literature, are available to define dependencies between requirements themselves, such as refinements, alternatives, conflicts, implications or impacts. Requirements may also be fulfilled by existing interfaces or require complex structural modifications. All these decisions may be further documented by **design rationale** to detail the reasons behind a decision, like its strengths, weaknesses, hypotheses or constraints.

RQ3.1 How can we use model transformation techniques to iteratively refine SA models with new concerns ?

From the strong binding and integration of software architectures and architecturally-significant requirements, we defined an **architectural framework** where every modification in an architecture representation is specified in terms of formal model transformations. For many structural modifications, like moving components from one composite to another, the existing links are automatically updated in order to speed up the work of software architects. The complete history of model modifications is explicitly kept in memory as part of the architectural knowledge and also

for rollback reasons or design alternative explorations. Moreover, as an iteration is recorded as a set of transformations belonging to a design decision, its rationale may also be explicitly documented in the requirement listing for latter references.

RQ3.2 How can we use model transformation techniques to build and document reusable architectural patterns ?

Patterns are directly specified as model transformations to be easily *instantiated* in architectural models. Furthermore, since a set of transformations is always part of a decision, patterns are directly documented with the same requirement formalism, as independent solutions, to recurrent problems expressed as requirements. The requirement listing offers a semi-formal mechanism that also suits to depict the target problem that a pattern solves, its advantages and limitations, as well as its hypotheses and constraints under which it is usable. But, as discussed in its dedicated chapter, it is possible that all structural aspects of a particular pattern may not be expressed as automatic transformations.

pIck One, Document And tranSform Strategy

All these contributions can be summarized into our **IODASS** framework, where architects are able to document and trace model evolutions **structurally**, through the iterative transformation process, and **wisely**, through the design rationale and relationships between decisions concerning the architecturally significant requirements. As partially observed during the empirical study we conducted, we believe our rigorous design process, but somewhat lightweight in its implementation, may help architects to keep control over a software architecture on the long run.

Identified Limitations

Obviously, the present research is not free of limitations. We cross over a list of identified shortcomings for our approach.

An integrated framework

All languages of the framework are closely integrated to each others. This holistic approach makes it powerful, but may be seen as an obstacle for many people. Even if some parts are optional and a built-in library is available with basic reusable constructs, the expressiveness of the architectural language makes it a bit complex at first sight. Many modeling elements exist, allowing to specify in a detailed way an architecture, but this level of specification is not always required by engineers, especially for early draft architectures.

The stringent modification process may also appear discouraging, as we observed for one team during our evaluation case study. We designed the framework this way because we believe in the systematic definition of model modifications, even if this zealous way of doing is also a drawback for very fast changes.

The particularly rich semantics of the transformation rules may also appear discouraging at first glance. We defined six main operators, which may look quite a lot. We decided to provide a semantically rigorous language to ease the work of architects through automatic updates and enable fine-grained modifications to models, but this would require a bit of effort at first.

Textual syntax only

Our framework currently relies on textual editors only. The syntax has been designed as relatively verbose and declarative in purpose to be readable by non practitioners, thought we did not explicitly evaluated the syntax itself. Even if for model merging (textual *line-by-line* verifications) or completeness reasons, textual models are often preferable, they have been seen as less intuitive during our validation phase. As we will describe later on, a combined approach with graphical visualization and possibly model editing too, should be investigated.

Datatype inheritance

In our architectural language, datatypes, namely `GenericTypes`, may not be extended. Type inheritance should be integrated in the language, in combination to the flexible composition mechanism. This would require to modify the formalization of the duck-typing composition accordingly, to take into considerations the subtypes of a given datatype.

No industrial case study

The comparative case study was rather small and conducted in a controlled environment. Ideally, a larger case study should be conducted, eventually with professionals, to identify the benefits of our approach. Such a larger experiment could be conducted in a comparable manner, with a control group using either SysML or another set of languages and tools dedicated to structural and requirement modeling with the same assets as the ones we were interested in, as defined in the protocol.

Another possibility would be to conduct a retrospective case study over an industrial software application and compare both results. But this technique would require to know the approximative amount of time spent to develop the existing solution in order to also evaluate the exceeding work required to develop the same application under our framework, if any. Thought, comparable profiles for modelers and/or analysts than the ones that develop the industrial software should participate in the study to avoid heterogeneity biases.

Last, more focused validations could also be conducted to evaluate the readability, as well as the level of structural details of produced models with time-framed focus group and external reviewers. With smaller, but more detailed and specific requirements asking for the evaluation of many alternatives or a deep knowledge in software architecture design, the expected benefits in rationale documentation could be further evaluated.

Java template generator

The developed code generator is limited to Java source code and creates relatively limited templates with method signatures that can be extended by concrete implementations. Even if this way of doing is common to separate business classes from generated classes, as we will discuss in the next section, a round-trip code management would be an asset. Also, for now, only components and interfaces are generated from model elements. The generation of protocol-specific connection facilities should be investigated in order to, for example, create proxy or *middleware-oriented* classes that will be used to interconnect concrete implementation classes.

Research Perspectives

To conclude this dissertation, we highlight interesting perspectives for possible future research.

Combined textual-graphical editors

The combination of a textual and a graphical modeling environment should be investigated. As suggested by many participants to the validation phase and the research community, at least a visualization facility should be implemented. Since recently, new technologies, some of them based on the EMF framework, have appeared and offer flexible and fast-delivery environments to build highly configurable visualization tools as Eclipse plugins. We particularly note the *GraphViz*[8] open-source initiative that allows to represent any graph-based structures either in standalone tools or inside Eclipse. The Eclipse *Graphical Editing Framework*[9] offers the possibility to build graphical editors inside the eclipse ecosystem. This would be a more complete, thought more complex and time-consuming, solution.

Another strategy would be to hide the textual transformation language behind the tracing of graphical modifications performed by the modeler. We guess this would probably require another Ph.D, but we think of a graphical environment where formal transformations are recorded as the modeler applies modifications onto a software architectural representation. At the end of his task, a rationalization process would transform all traces into macro-transformations that will serve as model deltas, just like in our framework, but completely transparent to the modeler. Still, he would be able to write its own transformations that could be applied graphically for instant visualization. Likewise, he would be able to inject or identify architectural patterns, again graphically.

Viewpoint-based filters

Combined to a graphical view, a facility to define custom viewpoints could be considered. With such a visualization, one may be able to abstract part of the system based on either some components (s)he is interested in, based on a list of requirements

[8] http://www.graphviz.org/
[9] http://www.eclipse.org/gef/

or on specific user-defined properties. In case of large system architectures, these additional views would be particularly valuable to help architects to concentrate on specific parts of the system, without crawling over the overall model when their interested in some specific features or properties.

Round-trip generation support and release management

One of the code generator's *Holy Grail* is the round-trip management of model and code. Since architecture evolutions are traced as formal model transformations, a complete workflow could be created to re-generate code from modifications of a particular model. The other way around, as we separated the generated code from its concrete implementation, code observer mechanisms could be developed to identify model violations or semantic refinements of some classes and warn the modeling team accordingly.

The proposed history tracing is relatively simple and lack of summary or graphical visualization facilities to identify deltas between versions at a glance. One could investigate the possibility to identify and document *releases* at some point of the iteration process. In case of a round-trip support, macro-transformations between releases could then been isolated to specify evolutions or patches that could be applied semi-automatically to existing software.

Property analysis, behavioral specifications and verifications

The user-defined property mechanism currently allows to annotate modeling elements and decisions rationale to refine their semantics. We presented some consistency checks one may perform on them when dealing with design rationale. A valuable extension of our approach would be to add model analysis based on these user-defined values, *e.g.* does the required disk space for a piece of software is provided by its deployment target or if a certain needed response time may be ensured with the concrete communication medium.

Also, this mechanism could be extended to allow the definition of behavioral specifications. The structured property constructs may already be used to specify static behavioral specifications. An appropriate model verification tool could be integrated into our tool suite to verify the presence or absence of some properties or the definition of conflicting values of the same property, in a larger scope than the one we defined between rationale constructs. Alternatively, relations between correlated properties could be integrated too in order to, for example, ensure a greedy software is deployed on a sufficiently powerful node, or a heavily demanding service in terms of the expected throughput may effectively have access to an adequate communication medium.

Inter-dependencies of services could also be added into the framework to represent the sequence of services activations for a specific use case. More generally, pre/postconditions could be integrated into the structural language, also as extensions of the property language, to annotate the signatures of services regarding the expected states of their parameters, for example.

APPENDIX A

DAD-PROPERTY XTEXT GRAMMAR

```
grammar be.unamur.info.iodass.textual.property.Property with org.eclipse.xtext.common.
    Terminals

import "platform:/resource/be.unamur.info.iodass.model/model/Property"
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

PropertyModel :
  'package' pname=FQN ('['revision' revision=REVNUMBER']')?';'
  properties += Property*
  'dadproperties' name=ID '{'
    elements += Element*
  '}';

Element :
  PropertyGroup | PropertyDeclaration | EnumDeclaration;

PropertyGroup returns PropertyGroup :
  'group for' target=PropertyTarget '{'
    (references+=[PropertyStatement|FQN]';' | properties += PropertyStatement)*
  '}';

PropertyType returns PropertyType:
  BooleanType | IntegerType | DecimalType | StringType | EnumType;

PropertyDeclaration returns PropertyDeclaration:
  'property for' target=PropertyTarget property=PropertyStatement;

PropertyStatement returns PropertyStatement :
  name=ID '{'
    'type' type=PropertyType (strictly?='strictly')? (ordering=OrderingLiteral)?';'
    ('unit' unit=STRING';')?
    ('semantics' semantics=STRING';')?
  '}';

EnumDeclaration returns EnumDeclaration:
  'enum' name=ID '{'
    literals+=EnumLiteral ( "," literals+=EnumLiteral)*
  '}';
```

```
38
39 EnumLiteral :
40   {EnumLiteral} name=ID;
41
42 BooleanType :
43   {BooleanType} 'boolean' ;
44
45 IntegerType :
46   {IntegerType} 'int' ;
47
48 DecimalType :
49   {DecimalType} 'decimal' ;
50
51 StringType :
52   {StringType}'string' ;
53
54 EnumType returns EnumType:
55   name=[EnumDeclaration|FQN];
56
57 //
58 // PROPERTIES
59 // must be redefined in DAD model too because DAD grammar does not inherit from property
       (unable to extend twice)
60 //
61
62 Property returns Property:
63   name=[PropertyStatement|FQN] ':' (value=Value |   evalue=[EnumLiteral|FQN])';';
64
65 Value returns Value:
66   {StringLiteral} value=STRING
67   | {BooleanLiteral} value=BOOLEANVALUE
68   | {IntegerLiteral} value=INT
69   | {DecimalLiteral} value=DECIMALVALUE ;
70
71 REVNUMBER :
72   INT('.'INT)*;
73
74 DECIMALVALUE returns ecore::EBigDecimal:
75   INT'.'INT;
76
77 terminal BOOLEANVALUE returns ecore::EBoolean:
78   'true' | 'false';
79
80 enum PropertyTarget returns PropertyTarget:
81   componenttype | interface | linktype | protocol | nodetype | gatetype | mediumtype |
       service  | model | asr
82   | one2one | one2many | many2many | simple | random | broadcast | loadbalancing;
83
84 enum OrderingLiteral returns OrderingLiteral:
85   asc | desc ;
86
87 FQN :
88   ID('.'ID)*;
```

APPENDIX B

DAD XTEXT GRAMMAR

```
grammar be.unamur.info.iodass.textual.archi.Dad with org.eclipse.xtext.common.Terminals

import "platform:/resource/be.unamur.info.iodass.model/model/Dad"
import "platform:/resource/be.unamur.info.iodass.model/model/Property" as property
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
import "http://www.eclipse.org/xtext/common/JavaVMTypes" as jvmTypes

DADModel returns DADModel:
  {DADModel}
  'package' pname=FQN ('['revision' revision=REVNUMBER']')? ';'
  imports+=Import*
  properties+=Property*
  'dadmodel' name=ID '{'
    (definition=Definition)?
    (assemblage=Assemblage)?
    (deployment=Deployment)?
  '}';

Import returns property::Import:
  {property::Import}
  'import' importedNamespace=FQNWithWildCard';';

//
// 1.Definition
//

Definition returns Definition:
  {Definition}
  'definition' '{'
    constructs += ConstructType*
    dependencies += Dependency*
  '}';

ConstructType returns ConstructType:
  GenericType | ComponentType | Protocol | LinkType | NodeType | MediumType | GateType;

Dependency returns Dependency:
  LinkageType | LeakUsage;
```

```
39
40 // data types
41
42 GenericType returns GenericType:
43   DataStructure | Interface | Primitive;
44
45
46 DataStructure returns DataStructure:
47   {DataStructure}
48   'struct' name=ID (many?='[]')? '{'
49     (dataFields+=DataField)*
50   '}';
51
52 DataField returns DataField:
53   {DataField}
54   fieldtype=[GenericType|FQN] (many?='[]')? name=ID';';
55
56 Primitive returns Primitive:
57   'datatype' name=ID ('mapped-to' javatype=[jvmTypes::JvmType|FQN])?';';
58
59
60 // Interfaces
61
62 Interface returns Interface:
63   {Interface}
64   'interface' name=ID '{'
65     (services += Service)*
66     ('properties' '{'
67       properties+=Property+
68     '}')?
69   '}';
70
71 Service returns Service:
72   (SyncOperation | AsyncOperation | Event | Exception | Flow) (properties+=Property)?';'
         ;
73
74 SyncOperation returns SyncOperation:
75   {SyncOperation}
76   'sync' resultType=[GenericType|FQN] (many?='[]')? name=ID'('(arglist+=Argument (","
       arglist+=Argument)*)?')' ('raises' raises+=[Exception|FQN] ("," raises+=[
       Exception|FQN])* )?;
77
78 AsyncOperation returns AsyncOperation:
79   {AsyncOperation}
80   'async' name=ID '('(arglist+=Argument ("," arglist+=Argument)*)?')';
81
82 Event returns Event:
83   {Event}
84   'event'  name=ID '('(arglist+=Argument (',' arglist+=Argument)*)?')';
85
86 Exception returns Exception:
87   {Exception}
88   'exception' name=ID '('(arglist+=Argument (',' arglist+=Argument)*)?')';
89
90 Flow returns Flow:
91   {Flow}
92   'flow' name=ID argument=Argument;
93
94 Argument returns Argument:
95   (access=ArgumentAccess)? argType=[GenericType|FQN] (many?='[]')? name=ID;
96
97
98 // component types, facets, usages and linkage types
99
100 ComponentType returns ComponentType:
101   {ComponentType}
```

```
102   (final?='final')? 'componenttype' name=ID ('extends' extends+=[ComponentType|FQN] (','
          extends+=[ComponentType|FQN])*)? '{'
103     (innerCompTypes += ComponentType
104     | innerReferences += [ComponentType|FQN]
105     | facets += Facet
106     | innerConfig += Dependency)*
107   '}';
108
109 Facet returns Facet:
110   {Facet}
111   usedType=FacetUsedType typeOfFacet=[Interface|FQN] 'as' name=ID';';
112
113 LinkageType returns LinkageType:
114   {LinkageType}
115    'linkagetype from' fromFacet=[Facet|FQN] 'to' toFacet=[Facet|FQN] 'with' linkType=[
          LinkType|FQN]';';
116
117 LeakUsage returns LeakUsage:
118   {LeakUsage}
119   'usage from' fromCompType=[ComponentType|FQN] 'to' toCompType=[ComponentType|FQN]';';
120
121
122 // linktypes and protocol
123
124 LinkType returns LinkType:
125   ConnectorType | DelegateType;
126
127 ConnectorType returns ConnectorType:
128   {ConnectorType}
129   'connectortype' name=ID ('extends' extends=[LinkType|FQN])?'{'
130     'mode' mode=ConnectionMode ';'
131     properties+=Property*
132     ('accepts' protocolList+=[Protocol|FQN] ("," protocolList+=[Protocol|FQN])*';' )?
133   '}';
134
135 DelegateType returns DelegateType:
136   {DelegateType}
137   'delegatetype' name=ID ('extends' extends=[LinkType|FQN])? '{'
138     'mode' mode=DelegationMode';'
139     properties+=Property*
140     ('accepts' protocolList+=[Protocol|FQN] ("," protocolList+=[Protocol|FQN])*';')?
141   '}';
142
143
144 Protocol returns Protocol:
145   {Protocol}
146   'protocol' name=ID ('extends' extends=[Protocol|FQN])? '{'
147     'layer' layer=CommunicationLayer';'
148     properties+=Property*
149   '}';
150
151
152 // hardware types
153
154 NodeType returns NodeType:
155   {NodeType}
156   'nodetype' name=ID ('extends' extends=[NodeType|FQN])? '{'
157     (gates+=Gate)*
158     (properties+=Property)*
159   '}';
160
161 MediumType returns MediumType:
162   {MediumType}
163   'mediumtype' name=ID  ('extends' extends=[MediumType|FQN])? '{'
164     ('supports' protocolList+=[Protocol|FQN] (',' protocolList+=[Protocol|FQN])*';')?
165     properties+=Property*
166   '}';
```

```
167
168 GateType returns GateType:
169   {GateType}
170   'gatetype' name=ID ('extends' extends=[GateType|FQN])?'{'
171     'supports' protocolList+=[Protocol|FQN] (',' protocolList+=[Protocol|FQN])*';'
172     properties+=Property*
173   '}';
174
175 Gate returns Gate:
176   {Gate}
177   typeOfGate=[GateType|FQN] ('[' size=INT ']')? name=ID';';
178
179 //
180 // 2.Assemblage
181 //
182
183 Assemblage returns Assemblage:
184   {Assemblage}
185   'assemblage' '{'
186     sois += SoI*
187     linkages+=Linkage*
188   '}';
189
190 SoI returns SoI:
191   {SoI}
192   'soi' name=ID ('['mincard=INT maxcard=INT']')? ':' typeOfSoi=[ComponentType|FQN] (
          bigbang?='bigbang')? '{'
193     (exposes+=Port)*
194     ('creates' creates+=[SoI|FQN] (',' creates+=[SoI|FQN])*';')?
195     ('destroys' destroys+=[SoI|FQN] (',' destroys+=[SoI|FQN])*';')?
196     (properties+=Property)*
197   '}';
198
199 Port returns Port:
200   {Port}
201   typeOfPort=[Facet|FQN] ('[' size=INT ']')? 'as' name=ID 'on' protocol=[Protocol|FQN]';
          ' ;
202
203
204 Linkage returns Linkage:
205   {Linkage}
206   'linkage from' fromPort=[Port|FQN] ('['fromLow=INT (fromUp=INT)?']')? 'to' toPort=[
          Port|FQN] ('['toLow=INT (toUp=INT)?']')? 'with' linktype=[LinkType|FQN]';';
207
208
209 //
210 // 3.Deployment
211 //
212
213 Deployment returns Deployment:
214   {Deployment}
215   'deployment' '{'
216     (nodes += Node
217     | sites+=Site
218     | plugs+=PlugCable
219     | deploys+=Deploy
220     | openings +=Open)*
221   '}';
222
223 Node returns Node:
224   {Node}
225   'node' name=ID ('[' size=INT ']')? ':' typeOfNode=[NodeType|FQN]';';
226
227 Site returns Site:
228   {Site}
229   'site' name=ID '{'
230     'situation' situation=STRING';'
```

```
231     (properties+=Property)*
232     'contains' nodes+=[Node|FQN] (',' nodes+=[Node|FQN])*';'
233   '}';
234
235 Deploy returns Deploy:
236   {Deploy}
237   'deploy' soi=[SoI|FQN] ('['soiFromId=INT  (soiToId=INT)?']')? 'on' node=[Node] ('['
          fromNodeId=INT  (toNodeId=INT)?']')?';';
238
239 PlugCable returns PlugCable:
240   {PlugCable}
241   'plug' mediumType=[MediumType|FQN] 'from' fromNode=[Node|FQN] ('['fromNodeMinId=INT  (
          fromNodeMaxId=INT)?']')?'::'fromGate=[Gate|FQN] ('['fromGateId=INT']')? 'to'
          toNode=[Node|FQN] ('['toNodeMinId=INT  (toNodeMaxId=INT)?']')?'::'toGate=[Gate|
          FQN] ('['toGateId=INT']')?';';
242
243 Open returns Open:
244   {Open}
245   'open' port=[Port|FQN] ('['fromPortId=INT (toPortId=INT)?']')? 'on' onNode=[Node|FQN]
          ('['fromNodeId=INT  (toNodeId=INT)?']')?'::'gate=[Gate|FQN] ('['gateId=INT']')?';
          ';
246
247 //
248 // PROPERTIES (redundant with rule in property MM, but can't inherit twice in xtext
249 //
250
251 Property returns property::Property:
252   name=[property::PropertyStatement|FQN] ':' (value=Value | evalue=[property::
          EnumLiteral|FQN])';';
253
254 Value returns property::Value:
255   {property::StringLiteral} value=STRING
256   | {property::BooleanLiteral} value=BOOLEANVALUE
257   | {property::IntegerLiteral} value=INT
258   | {property::DecimalLiteral} value=DECIMALVALUE ;
259
260
261 //
262 // Types
263 //
264
265 terminal BOOLEANVALUE returns ecore::EBoolean:
266   'true' | 'false';
267
268 DECIMALVALUE returns ecore::EBigDecimal:
269   INT'.'INT;
270
271 //
272 // Technical rules, enums and terminal
273 //
274
275 FQN :
276   ID('.'ID)*;
277
278 FQNWithWildCard:
279   FQN '.*'?;
280
281 REVNUMBER :
282   INT('.'INT)*;
283
284 enum ArgumentAccess returns ArgumentAccess:
285   in='in' | out='out' | inout='inout';
286
287 enum FacetUsedType returns FacetUsedType:
288   implements='implements' | uses='uses';
289
```

```
290 enum CommunicationLayer returns CommunicationLayer: /* undefined is hidden from content
        assist */
291   physical='physical' | datalink='datalink' | network='network' | transport='transport'
          | session='session' | presentation='presentation' | application='application' |
          samespace='samespace';
292
293 enum ConnectionMode returns ConnectionMode:
294   one2one | one2many | many2many;
295
296 enum DelegationMode returns DelegationMode:
297   simple | random | broadcast | loadbalancing;
```

APPENDIX C

ASR XTEXT GRAMMAR

```
grammar be.unamur.info.iodass.textual.asr.Asr with be.unamur.info.iodass.textual.archi.Dad

import "platform:/resource/be.unamur.info.iodass.model/model/Asr"
import "platform:/resource/be.unamur.info.iodass.model/model/Dad" as dad
import "http://www.eclipse.org/emf/2002/Ecore" as ecore

ASRModel returns ASRModel:
  {ASRModel}
  'package' pname=FQN ('['revision' revision=REVNUMBER']')?';'
  imports+= Import*
  properties+=Property*
  'asrmodel' name=ID ('with' importedNamespace=FQN)? '{'
    asr+=ASR+
  '}';

ASR returns ASR:
  {ASR}
  (final?='final')? asrType=ASRType name=ID 'assigned' isInCharge=[dad::ConstructType|FQN] '{'
    'description' longDesc=STRING';'
    ('priority' priority=INT';')?
    (properties+=Property*)?
    (decisions+=Decision)*
  '}';

Decision:
  (DDImplemOrUsage // assign to interface
  | DDAlternative // refinement (may be an alternative)
  | DDAssignment  // reassign to other construct
  | DDImplication // asr implies selection of other asr
  | DDExclude // mutual exclusion between 2 asr
  | DDImpact // negative, neutral or positive impact between requirement
  | DDRealisation) '{' // transformation
    (rationale+=Rationale)+
  '}';

DDImplemOrUsage returns DDImplemOrUsage:
```

```
37    {DDImplemOrUsage} usedType=FacetUsedType interface=[dad::Interface|FQN] ;
38
39  DDAlternative returns DDAlternative:
40    {DDAlternative} 'refines' asr=[ASR|FQN] ('is' altType=DDAlternativeType)?;
41
42  DDAssignment returns DDAssignment:
43    {DDAssignment} 'reassigned' construct=[dad::ConstructType|FQN] ;
44
45  DDExclude returns DDExclude:
46    {DDExclude} 'exclude' asr=[ASR|FQN];
47
48  DDImpact returns DDImpact:
49    {DDImpact} 'impact' (type=ImpactType)? asr=[ASR|FQN];
50
51  DDImplication returns DDImplication:
52    {DDImplication} 'implies' asr=[ASR|FQN];
53
54  DDRealisation returns DDRealisation:
55    {DDRealisation} 'realisation' transformation=FQN;
56
57  Rationale returns Rationale:
58    {Rationale} type=RationaleType (property=Property)? description=STRING';';
59
60  /*
61   * ENUMS
62   */
63
64  enum ASRType :
65    func | nonfunc;
66
67  enum DDAlternativeType :
68    alternative | selected ;
69
70  enum RationaleType :
71    assessment | assumption | strength | weakness | constraint;
72
73  enum ImpactType :
74    neutrally | negatively | positively;
```

APPENDIX D

DAD-T XTEXT GRAMMAR

```
grammar be.unamur.info.iodass.textual.transfo.DadT with be.unamur.info.iodass.textual.archi.Dad

import "platform:/resource/be.unamur.info.iodass.model/model/DadT"
import "platform:/resource/be.unamur.info.iodass.model/model/Dad" as dad
import "platform:/resource/be.unamur.info.iodass.model/model/Asr" as asr

DADTModel returns DADTModel:
{DADTModel}
  'package' pname=FQN ('['revision' revision=REVNUMBER']')?';'
  imports += Import*
  'asrmodel' asrmodel=FQN';'
  ('dadmodel' dadmodel=FQN';')?
  properties+=Property*
  'transformationset' name=ID 'concerns' asr+=[asr::ASR|FQN] (',' asr+=[asr::ASR|FQN])* '{'
    (transfo+=Transformation)+
  '}';

Transformation :
  CreateStatement | DeleteStatement | AlterStatement
  | ReplaceConstruct | MoveConstruct | RenameConstruct
  | IncarnateConstruct | DeployConstruct
  | IncludeStatement;


IncludeStatement returns IncludeStatement:
  'include' importedNamespace=FQN';';

//
// Creation
//

CreateStatement :
  'create' (CreateConstruct | CreateLinkageType | CreateUsage);

CreateConstruct returns CreateConstruct:
  construct=ConstructType ('parent' parentCT=[dad::ComponentType|FQN]';')?;
```

```
37
38 CreateLinkageType returns CreateLinkageType:
39   linkageType = LinkageType;
40
41 CreateUsage returns CreateUsage:
42   leakUsage=LeakUsage;
43
44
45 //
46 // deletion
47 //
48
49 DeleteStatement:
50   'delete' (DeleteConstruct | DeleteLinkageType | DeleteUsage | DeleteFacet | DeleteGate
51     | DeletePort | DeleteSoI | DeleteLinkage
52     | DeleteNode | DeleteDeploy | DeleteOpen | DeletePlug);
53
54 DeleteConstruct returns DeleteConstruct:
55   construct=[dad::ConstructType|FQN] ';';
56
57 DeleteLinkageType returns DeleteLinkageType:
58   linkageType=LinkageType;
59
60 DeleteUsage returns DeleteUsage:
61   usage=LeakUsage;
62
63 DeleteFacet returns DeleteFacet:
64   'facet' facet=[dad::Facet|FQN] ';';
65
66 DeletePort returns DeletePort:
67   'port' port=[dad::Port|FQN] ';';
68
69 DeleteGate returns DeleteGate:
70   'gate' gate=[dad::Gate|FQN] ';';
71
72 DeleteSoI returns DeleteSoI:
73   'soi' soi=[dad::SoI|FQN] ';';
74
75 DeleteLinkage returns DeleteLinkage:
76   linkage=Linkage;
77
78 DeleteNode returns DeleteNode:
79   'node' node=[dad::Node|FQN]';';
80
81 DeleteDeploy returns DeleteDeploy:
82   deploy= Deploy;
83
84 DeletePlug returns DeletePlug:
85   plug=PlugCable;
86
87 DeleteOpen returns DeleteOpen:
88   open=Open;
89
90
91 //
92 // alterations
93 //
94
95 AlterStatement returns AlterStatement:
96   'alter' (AlterInterface | AlterDataStructure | AlterProtocolList | AddGate | AddFacet
       | AlterSoI  | AlterSite);
97
98
99 // interfaces
100
101 AlterInterface returns AlterInterface:
102   'interface' interface=[dad::Interface|FQN] '{'
```

```
103     (alterations+=InterfaceAlteration)+
104   '}';
105
106 InterfaceAlteration returns InterfaceAlteration:
107   CreateService | DeleteService | RewriteService;
108
109 CreateService returns CreateService:
110   'add' service=Service;
111
112 DeleteService returns DeleteService:
113   'remove' service=[dad::Service|FQN]';';
114
115 RewriteService returns RewriteService:
116   'rewrite' service=[dad::Service|FQN] ( 'by' rewriting=Service | '{' alterArguments +=
          AlterArgument+ '}');
117
118 AlterArgument returns AlterArgument:
119   CreateArgument | RewriteArgument | DeleteArgument;
120
121 CreateArgument returns CreateArgument:
122   'add' argument=Argument';';
123
124 DeleteArgument returns DeleteArgument:
125   'remove' argument=[dad::Argument|FQN]';';
126
127 RewriteArgument returns RewriteArgument:
128   'replace' argument=[dad::Argument|FQN] 'by' rewriting=Argument';';
129
130 AlterDataStructure returns AlterDataStructure:
131   'struct' dataStructure=[dad::DataStructure|FQN] '{'
132     (alterations+=DataStructureAlteration)+
133   '}';
134
135
136 // data structures
137
138 DataStructureAlteration returns DataStructureAlteration:
139   CreateField | RewriteField | DeleteField;
140
141 CreateField returns CreateField:
142   'add' dataField=DataField;
143
144 RewriteField returns RewriteField:
145   'replace' dataField=[dad::DataField|FQN] 'by' rewriting=DataField;
146
147 DeleteField returns DeleteField:
148   'remove' dataField=[dad::DataField|FQN]';';
149
150
151 // protocols
152
153 AlterProtocolList returns AlterProtocolList:
154   (AlterProtocolConnectorType | AlterProtocolDelegationType | AlterProtocolGateType |
          AlterProtocolMediumType)
155   ('add' | deletion?='delete') protocols+=[dad::Protocol|FQN] (',' protocols+=[dad::
          Protocol|FQN])*';';
156
157 AlterProtocolConnectorType returns AlterProtocolConnectorType:
158   'connectortype' connectortype=[dad::ConnectorType|FQN] ;
159
160 AlterProtocolDelegationType returns AlterProtocolDelegationType:
161   'delegationtype' delegatetype=[dad::DelegateType|FQN] ;
162
163 AlterProtocolGateType returns AlterProtocolGateType:
164   'gatetype' gatetype=[dad::GateType|FQN] ;
165
166 AlterProtocolMediumType returns AlterProtocolMediumType:
```

```
167   'mediumtype' mediumtype=[dad::MediumType|FQN] ;
168
169 // facet, gate and port
170
171 AddFacet returns AddFacet:
172   'componenttype' componentType=[dad::ComponentType|FQN] '{'
173     (facets+=Facet)+
174   '}';
175
176 AddGate returns AddGate:
177    'nodetype' nodeType=[dad::NodeType|FQN] '{'
178     (gates+=Gate)+
179   '}';
180
181 AlterSoI returns AlterSoI:
182   'soi' soi=[dad::SoI|FQN] '{'
183     (ports+=Port)*
184     ('card' '[' mincard=INT maxcard=INT ']'';')?
185     ('creates' creates+=[dad::SoI|FQN] (',' creates+=[dad::SoI|FQN])*';')?
186     ('destroys' destroys+=[dad::SoI|FQN] (',' destroys+=[dad::SoI|FQN])*';')?
187     ('bigbang' bigbang=BOOLEANVALUE';')?
188   '}';
189
190 AlterSite returns AlterSite:
191   'site' site=[dad::Site|FQN] '{'
192     ('rename' newname=ID';')?
193     ('situation' newsituation=STRING ';')?
194     ('add' addnodes+=[dad::Node|FQN] (',' addnodes+=[dad::Node|FQN])*';')?
195     ('remove' remnodes+=[dad::Node|FQN] (',' remnodes+=[dad::Node|FQN])*';')?
196   '}';
197
198
199 //
200 // replacement, renaming and move
201 //
202
203 ReplaceConstruct returns ReplaceConstruct:
204   'replace' oldConstruct=[dad::ConstructType|FQN] 'by' newConstruct=[dad::ConstructType|
        FQN] (merge?='merge')? (keepOldName?='keepname')? ('overrides' '{' ovRules+=
        OverrideRule* '}')?';';
205
206 OverrideRule :
207   (OverrideComponentType | OverrideFacet | OverridePort) (keepOldName?='keepname')?';';
208
209 OverrideComponentType returns OverrideComponentType:
210   old=[dad::ComponentType|FQN] 'by' repl=[dad::ComponentType|FQN] (merge?='merge')? ;
211
212 OverrideFacet returns OverrideFacet:
213   'facet' old=[dad::Facet|FQN] 'by' repl=[dad::Facet|FQN];
214
215 OverridePort returns OverridePort:
216   'port' old=[dad::Port|FQN] 'by' repl=[dad::Port|FQN];
217
218 //
219 // move
220 //
221
222 MoveConstruct returns MoveConstruct:
223   'move' target=[dad::ConstructType|FQN] 'to' (newparent=[dad::ComponentType|FQN] | '
        root')';';
224
225 //
226 // rename
227 //
228
229 RenameConstruct returns RenameConstruct:
230   'rename' construct=[dad::ConstructType|FQN] 'as' newname=ID';';
```

```
231
232 //
233 // assemblage and deployment
234 //
235
236 IncarnateConstruct :
237   assemblage=Assemblage;
238
239 DeployConstruct :
240   deployment=Deployment;
```

APPENDIX E

IODASS TOOL SUITE USER GUIDE

E.1 Preamble

The **IODASS** plugin has been developed mainly as a *proof-of-concept* tool in order to test the feasibility of our transformation-centric framework. The majority of the exposed concepts regarding the domain specific languages and the design strategy have been implemented. We also tried to make it somewhat user friendly with file and project *wizards*, enhanced *outline* view, meaningful messages and a built-in sample.

We particularly paid attention to develop this plugin following some best practices, like documented and commented code, self explanatory naming conventions, separation of concerns between the different languages, and so forth.

Aside the language grammars reproduced in Appendix A to D, this guide is meant to help users to have an overview of the tool's architecture and how it places itself in the Eclipse ecosystem. We also describe the installation procedure from an Eclipse update site we set up. Afterwards, we present the built-in example dedicated to introduce the **IODASS** concepts in a simplified manner. We finally present the main

features of the tool with the various creation wizards, the transformation engine and the Java generator.

E.2 Xtext-Based Architectural Overview

As introduced in Chapter 6, the **IODASS** plugin relies on the Eclipse Modeling Framework[1], which is a basis for model-based development in the Eclipse platform. On top of EMF, many modeling tools have been created, like Papyrus[2] for UML diagrams, ATL transformations[3], Acceleo[4] textual (code) generator or Xtext[5] DSL development.

We found Xtext particularly suitable to our purpose, especially because this initiative is now well supported by the Itemis AG[6] company and a wide community of users. Since Xtext is itself a framework, we had to stick to its architecture, as shown in Figures 6.5 and 6.6. An Xtext 2 project is always divided at least in two distinct Eclipse projects, one for the language itself and one for the user interface. The language project follows this structure:

src	the source folder where the DSL developers put its own code
src-gen	the generated Java classes from the language grammar
xtend-gen	the generated Java classes from Xtend classes

In short, from an initial `MyGrammar.xtext` file, aside all generated classes to manipulate the grammar itself, the framework generates a set of folders and Java skeletons in the `src` folder, following this structure for the first project:

formatting	for *pretty-printing* purposes
generator	any code generator classes
naming	handles how the names of model elements must be created
scoping	manages the scope of a model elements (model imports)
serializer	customizes how models must be serialized
validation	custom model semantic validations

The UI project `src` folder is composed by the following packages:

contentassist	assists users when they write models
labeling	customizes the popup labels
outline	customizes the outline view
quickfix	applies quick-fixes to syntactical/semantical errors
wizard	specifies dedicated creation wizards

Four dedicated Xtext projects have been created, one per language, all following the same structures as described above. Furthermore, special handlers have been developed in committed `handler` packages to add the possibility to generate Java code templates from **DAD** models and run **DAD-T** transformation sets when clicking on the right model. Also, the transformation engine has been also placed into its own package, in the `DAD-T` language project.

[1] http://eclipse.org/emf/
[2] http://www.eclipse.org/modeling/mdt/papyrus/
[3] http://www.eclipse.org/atl
[4] http://www.eclipse.org/acceleo
[5] http://www.eclipse.org/Xtext
[6] http://www.itemis.com/

Finally, the language *"Xtext-compliant"* meta-models have been defined in a separated plugin, where the generated EMF code is also present.

E.3 Install Instructions

We created an Eclipse update site to ease the installation of the **IODASS** plugin at https://fabgilson.bitbucket.org/. We briefly summarize the installation procedure here. Thanks to the dependency resolver of Eclipse, one can start from any Eclipse distribution, but the *Java and DSL developers* and *Modeling tools* are the most appropriate with all needed dependencies already installed.

First, inside Eclipse, open the *Help* menu and select the *Install new software* option. A similar window as Figure E.1 will show up.

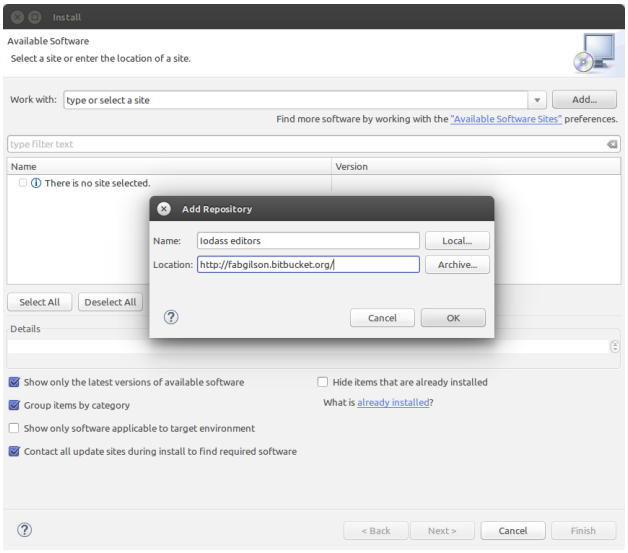


Figure E.1: Install new software window

After clicking the *Add* button on the top right of the window, fill in the aforementioned address for the **IODASS** update site with a label, like *Iodass editors* in the example, and click the *OK* button. The available plugin will be displayed and you can install it by ticking the checkbox right next to the plugin name and clicking the *Next* button, as shown in Figure E.2.

You will be presented with the user agreement that describes the license of the **IODASS** plugin. The plugin is distributed under the Eclipse Public License v1.0[7]. You may accept, click the *Finish* button and the plugin will be installed.

Last, you will have to restart Eclipse to make the changes effective and you will be able to create you first **IODASS** project.

[7] https://www.eclipse.org/legal/epl-v10.html

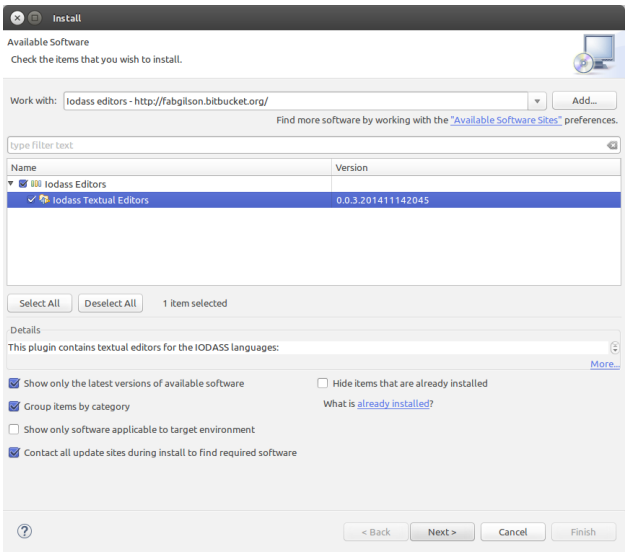


Figure E.2: Install **IODASS** editors

E.4 Create a New IODASS Project

In order to create a new **IODASS** project, we developed a creation wizard to ease the configuration and dependency setups. From the *File* menu, select the *New, Project...* options and search for the **IODASS** entry in the list of project types[8]. Figure E.3 depicts the **IODASS** entry in the project list.

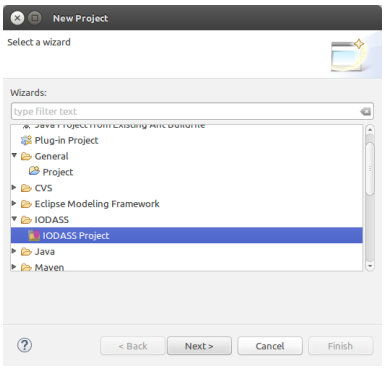


Figure E.3: New **IODASS** project wizard

After having filled in a name for your project, like *IodassTutorial* in our case, click the *Finish* button and the project will be shown in your package explorer. You should now be presented a similar window as depicted in Figure E.4.

[8]This can also be done from a right click in the *Package Explorer*

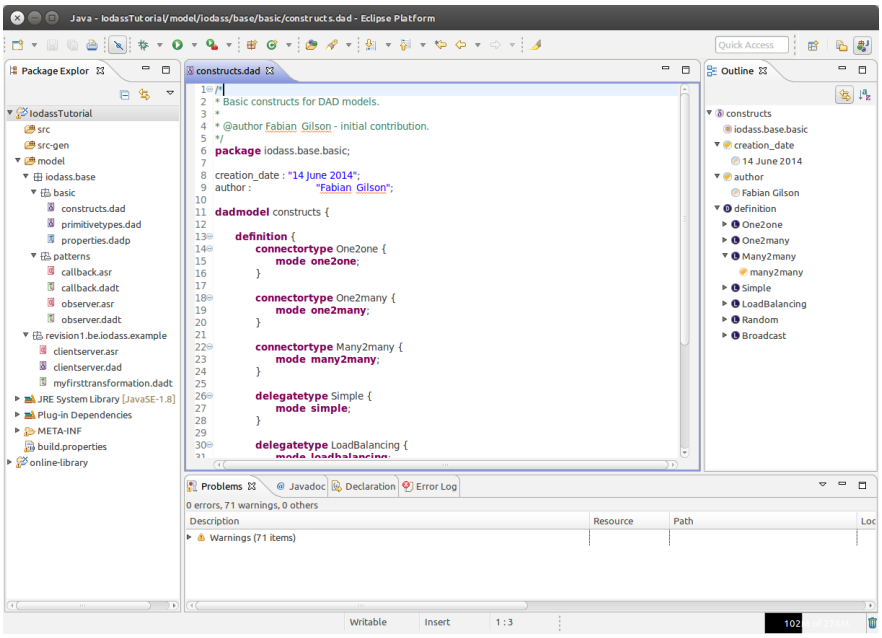


Figure E.4: New **IODASS** project

A new project always contains a set of built-in models with reusable constructs, properties and patterns in the `iodass.base` package. Also, a *helloworld* example is also provided to have a quick introduction to some types of constructs and to the transformation and generation engines. We will introduce both engines after having reviewed the content of the *helloworld* sample

E.5 Built-in Helloworld Example

A small client-server example is shipped with the plugin. It can be found in the `revision1.be.iodass.example` package. The **DAD** model is reproduced in the following code listing E.1.

```
1  package be.iodass.example [ revision 1 ] ;
2
3  dadmodel clientserver {
4    definition {
5      componenttype Client {}
6      componenttype Server {}
7
8      protocol TCP { layer transport; }
9
10     connectortype One2One {
11       mode one2one;
12       accepts TCP;
13     }
14
15     gatetype Ethernet { supports TCP; }
16
```

```
    nodetype Computer { Ethernet eth; }

    mediumtype RJ45 { supports TCP; }
  }

  assemblage {
    soi client : Client {}
    soi server : Server {}
  }

  deployment {
    node computer [2] : Computer;
    plug RJ45 from computer[0]::eth to computer[1]::eth;
    deploy client on computer[0];
    deploy server on computer[1];
  }
}
```

Listing E.1: DAD model of the `Client-Server` built-in sample

A dummy requirement is also expressed in the bound **ASR** model and this only requirement is implemented by the example transformation set, also part of the *helloworld* sample. We first give the details of the requirement in Listing E.2.

```
package be.iodass.example [ revision 1 ] ;

asrmodel clientserver with be.iodass.example.clientserver {
  // sample requirement definition
  func SayHello assigned Server {
    description "Say hello World!";
    realisation be.iodass.example.myfirsttransformation {
      assessment "functionality is trivial, an interface should make the trick.";
    }
  }
}
```

Listing E.2: AST listing of the `Client-Server` built-in sample

The DAD architectural model is enriched through model transformations, detailed in Listing E.3

```
package be.iodass.example [ revision 1 ] ;

asrmodel be.iodass.example.clientserver; // asr model where the requirement is defined
dadmodel be.iodass.example.clientserver; // linked dad model (to be transformed)

transformationset myfirsttransformation concerns SayHello {

  // create interface
  create interface Hello {
    sync void hello();
  }

  // create facets on each part
  alter componenttype Client{
    uses Hello as hello;
  }

  alter componenttype Server {
    implements Hello as hello;
  }

  // create ports for facets
  alter soi client {
```

```
24      Client.hello as hello on TCP;
25    }
26
27    alter soi server {
28      Server.hello as hello on TCP;
29    }
30
31    create linkagetype from Client.hello to Server.hello with One2One;
32
33    // connect client to server
34    assemblage {
35      linkage from client.hello to server.hello with One2One;
36    }
37
38    // open ports on client and server on ethernet gate
39    deployment {
40      open client.hello on computer[0]::eth;
41      open server.hello on computer[1]::eth;
42    }
43  }
```

Listing E.3: DAD-T set of the Client-Server built-in sample

Thought all these model samples are relatively trivial, they already illustrate a significant part of the available **IODASS** constructs. By right-clicking on the **DAD-T** file, we may select the *IODASS* menu and *Execute DAD transformation set*, as shown in Figure E.5.

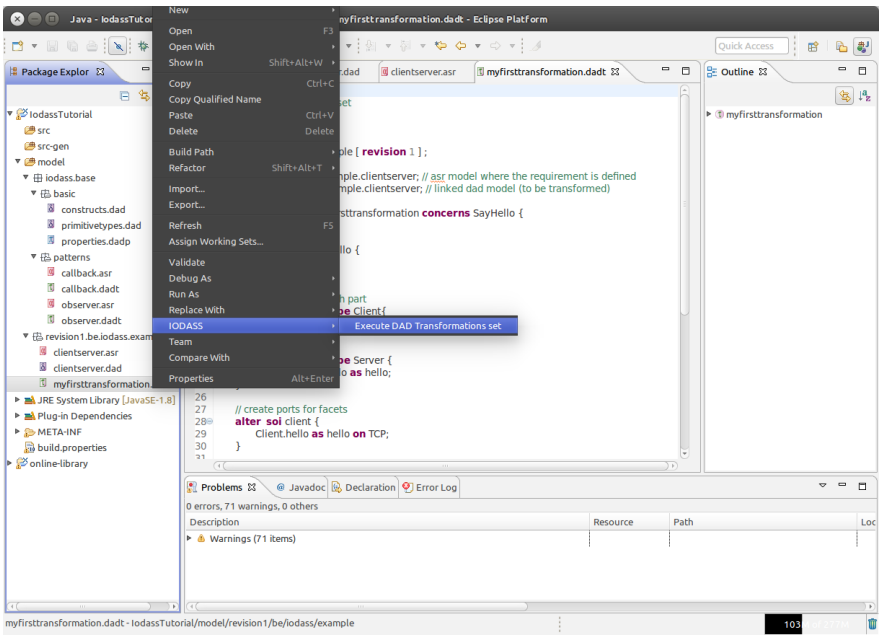


Figure E.5: Execute **DAD-T** set

E.6 Generate a New Iteration

After running the transformation, a new folder will be created with the updated **DAD** model together with a copy of the **ASR** model, as shown in Figure E.6.

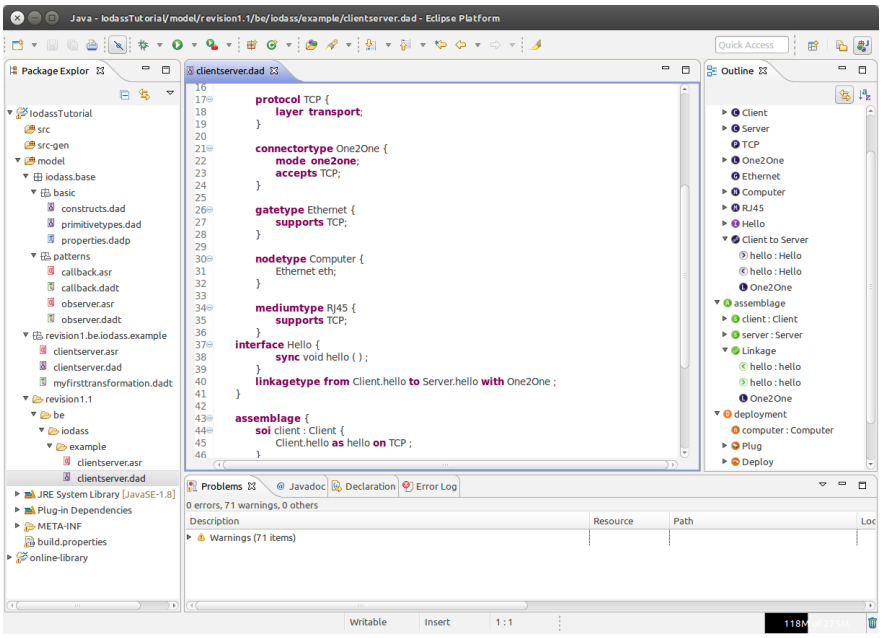


Figure E.6: New folder and model created from the execution of the **DAD-T** set

The `revision1.1` folder duplicates the package hierarchy of the `revision1` and we can continue refining our model iteratively, or backtrack to a previous version.

The `iodass.base` features are not duplicated unnecessarily and stay available from any revision number.

E.7 Generate Java Templates

We also have the possibility to generate Java template code from **DAD** models by right-clicking a **DAD** file, selecting the *IODASS* menu and click the *Generate Java code from DAD model* option, as illustrated in Figure E.7.

As a result, a set of Java resources are produced in the `src-gen` folder. Figure E.8 shows the content of the generated `Server` class and *Hello* interface. The generation timestamp and the revision number are also present in the header of the generated files to keep a link between the actual model they belong to.

A package referencing the `iodass.base` elements is also produced, but at current development time, it is empty. A similar package structure to the DAD model one is created, with all generated elements under the `clientserver` package.

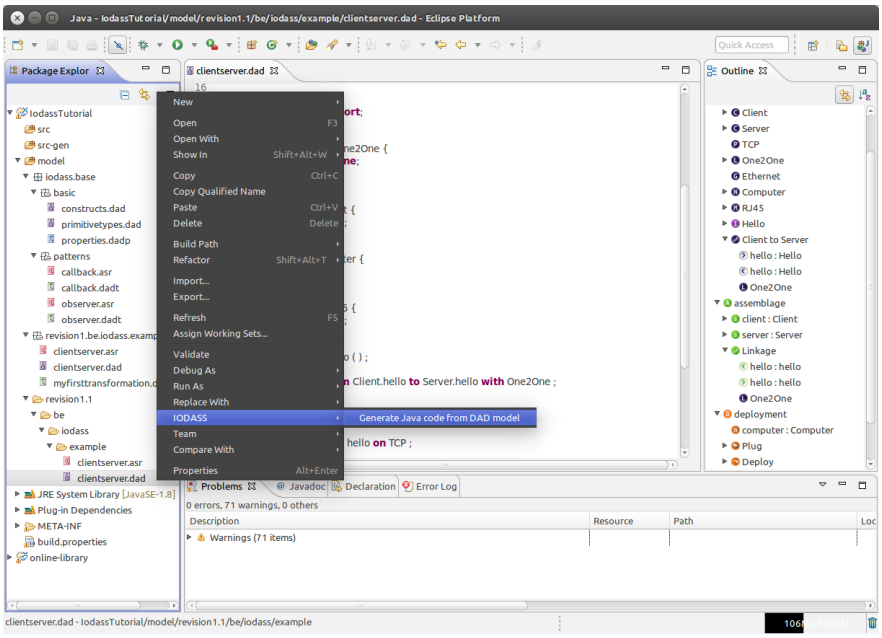


Figure E.7: Generate Java code from a **DAD** model

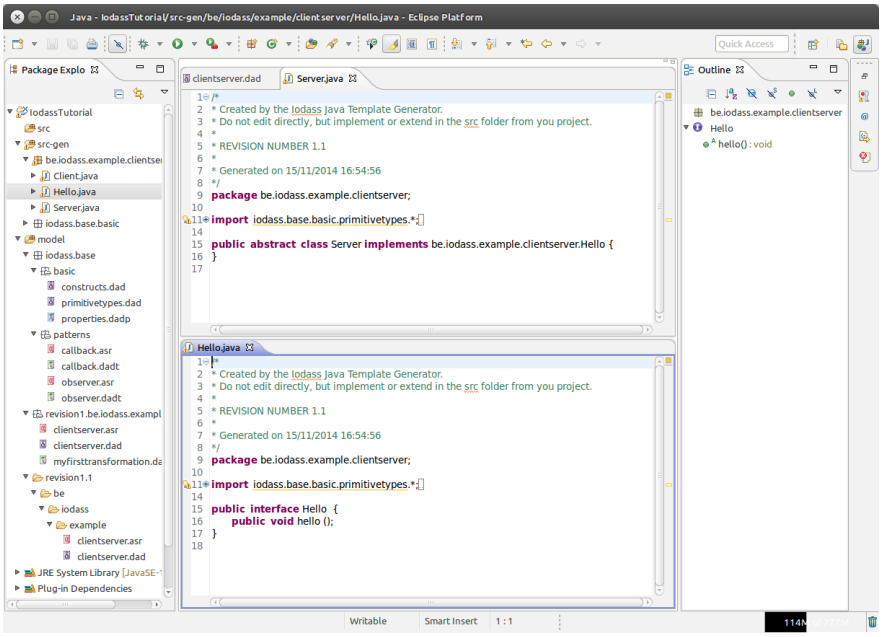


Figure E.8: Generated Java sources

Java developers may extend the generated class in the `src` folder to keep separated the generated and hand-written code.

E.8 Missing Features for Property Verifications

Regarding user properties, and specifically concerning their usages as `Rationale`, the various verifications of compatibility is not implemented as detailed in Section 4.4. At current development stage, all properties are considered as string values, even if they may be defined according to their declared types.

E.9 Concluding Remarks and Possible Enhancements

In this Appendix, we gave a user-centered overview of the **IODASS** plugin. We described the overall architecture within the Xtext framework and we detailed the installation procedure. We also described the various helps and built-in features of the plugin with many screenshots.

However, the plugin could be further extended with some features. First, even if we paid many attention to user feedbacks, in some cases a transformation may fail with few details on the reasons why the transformation actually flopped.

Second, the built-in library should be extended. A couple of basic construct types and properties are already specified, but some more effort could be put in that area, especially for architectural patterns.

Third, a wide range of syntactic and semantic validations are also performed, *e.g.* regarding the validity of `LinkageTypes`, name consistencies, etc. However, only a few quickfixes have been developed to *auto-magically* fix some recurrent errors.

Last, no detailed popup descriptions and documentation is actually available to help users to easily reference to their created constructs when editing a model.

BIBLIOGRAPHY

Aldrich, J., Chambers, C., and Notkin, D. (2002). Archjava: connecting software architecture to implementation. In **Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on**, pages 187–197.

Alexander, C. (1999). The origins of pattern theory: The future of the theory, and the generation of a living world. **IEEE Software**, 16(5):71–82.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Angel, S. (1977). **A Pattern Language - Towns, Buildings, Construction**. Oxford University Press.

Ali Babar, M., de Boer, R. C., Dingsoyr, T., and Farenhorst, R. (2007). Architectural knowledge management strategies: Approaches in research and industry. In **Proceedings of the 29th International Conference on Software Engineering Workshops**, ICSEW '07, pages 35–, Washington, DC, USA. IEEE Computer Society.

Ali Babar, M. and Gorton, I. (2007). A tool for managing software architecture knowledge. In **Proceedings of the Second Workshop on SHAring and Reusing Architectural Knowledge Architecture, Rationale, and Design Intent**, SHARK-ADI '07, pages 11–, Washington, DC, USA. IEEE Computer Society.

Ali Babar, M., Gorton, I., and Jeffery, R. (2005a). Capturing and using software architecture knowledge for architecture-based software development. In **Proceedings of the Fifth International Conference on Quality Software**, QSIC '05, pages 169–176, Washington, DC, USA. IEEE Computer Society.

Ali Babar, M. and Lago, P. (2009). Design decisions and design rationale in software architecture. **J. Syst. Softw.**, 82(8):1195 – 1197. SI: Architectural Decisions and Rationale.

Ali Babar, M., Northway, A., Gorton, I., Heuer, P., and Nguyen, T. (2008). Introducing tool support for managing architectural knowledge: An experience report. In **Proc. of the 15th Annual IEEE Int. Conf. and Workshop on the Engineering of Computer Based Systems (ECBS'08)**, pages 105 –113.

Ali Babar, M., Tang, A., Gorton, I., and Han, J. (2006). Industrial perspective on the usefulness of design rationale for software maintenance: A survey. In **Quality Software, 2006. QSIC 2006. Sixth International Conference on**, pages 201–208.

Ali Babar, M., Wang, X., and Gorton, I. (2005b). Pakme: A tool for capturing and using architecture design knowledge. In **Proc. of the 9th Int'l Multitopic Conference IEEE INMIC 2005**, pages 1–6.

Allen, R. (1997). **A Formal Approach to Software Architecture**. PhD thesis, School of Computer Science, Carnegie Mellon University. Issued as CMU Technical Report CMU-CS-97-144.

America, P., Rommes, E., and Obbink, H. (2004). Multi-view variation modeling for scenario analysis. In van der Linden, F., editor, **Software Product-Family Engineering**, volume 3014 of **Lecture Notes in Computer Science**, pages 44–65. Springer Berlin Heidelberg.

Amyot, D., Ghanavati, S., Horkoff, J., Mussbacher, G., Peyton, L., and Yu, E. (2010). Evaluating goal models within the goal-oriented requirement language. **Int. J. Intell. Syst.**, 25(8):841–877.

Atkinson, C. and Kühne, T. (2003). Model-driven development: A metamodeling foundation. **IEEE Softw.**, 20(5):36–41.

AUTOSAR (2013a). General specification of basic software modules, v1.1.0 r4.1 rev 2. Document ID 578: AUTOSAR_SWS_BSWGen.

AUTOSAR (2013b). Specification of standard types v1.5.0 r4.1 rev 2. Document ID 049: AUTOSAR_SWS_Standard.

Avgeriou, P., Kruchten, P., Lago, P., Grisham, P., and Perry, D. (2007). Architectural knowledge and rationale: Issues, trends, challenges. **SIGSOFT Softw. Eng. Notes**, 32(4):41–46.

Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A. J., Rutishauser, H., Samuelson, K., Vauquois, B., Wegstein, J. H., van Wijngaarden, A., and Woodger, M. (1960). Report on the algorithmic language algol 60. **Numerische Mathematik**, 2(1):106–136.

Backus, J. W., Beeber, R. J., Best, S., Goldberg, R., Haibt, L. M., Herrick, H. L., Nelson, R. A., Sayre, D., Sheridan, P. B., Stern, H., Ziller, I., Hughes, R. A., and Nutt, R. (1957). The FORTRAN automatic coding system. In **Western Joint Computer Conference: Techniques for Reliability**, IRE-AIEE-ACM '57 (Western), pages 188–198, New York, NY, USA. ACM.

Balasubramanian, D., Narayanan, A., van Buskirk, C. P., and Karsai, G. (2006). The graph rewriting and transformation language: Great. In **Proc. of the 3rd Int'l Workshop on Graph Based Tools (GraBaTs 2006)**, volume 1 of **Electronic Communications of the EASST**.

Barais, O. (2005). **Construire et Maîtriser l'évolution d'une architecture logicielle à base de composants**. PhD thesis, Laboratoire d'Informatique Fondamentale de Lille, Lille, France.

Barais, O., Duchien, L., and Meur, A.-F. L. (2005). A framework to specify incremental software architecture transformations. In **Proc. of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA 2005)**, pages 62–69. IEEE Computer Society.

Basili, V. R. (1992). Software modeling and measurement: The goal/question/metric paradigm. Technical report, University of Maryland at College Park, College Park, MD, USA.

Basili, V. R., Caldiera, G., and Rombach, H. D. (1994). Goal question metric approach. In **Encyclopedia of Software Engineering**, pages 528–532. John Wiley & Sons, Inc.

Basili, V. R. and Turner, A. J. (1975). Iterative enhancement: A practical technique for software development. **IEEE Trans. Softw. Eng.**, SE-1(4):390–396.

Bass, L., Clements, P., and Kazman, R. (2003). **Software Architecture in Practice, Second Edition**. Addison-Wesley Professional.

Beck, K. (1999). Embracing change with extreme programming. **Computer**, 32(10):70–77.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Manifesto for agile software development. retrieved Dec. 12, 2013.

Beck, K. and Boehm, B. (2003). Agility through discipline: a debate. **Computer**, 36(6):44–46.

Beedle, M., Devos, M., Sharon, Y., Schwaber, K., and Sutherland, J. (1999). Scrum: An extension pattern language for hyperproductive software development. **Pattern Languages of Program Design**, 4:637–651.

Benington, H. D. (1983). Production of large computer programs. **Annals of the History of Computing**, 5(4):350–361. originally available in Proc. ONR Symposium on Advanced Programming Methods for Digital Computers, June 1956, pp. 15–17.

Beugnard, A., Jézéquel, J.-M., Plouzeau, N., and Watkins, D. (1999). Making components contract aware. **Computer**, 32(7):38–45.

Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., and Lindow, A. (2006). Model transformations? transformation models! In **Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems**, MoDELS'06, pages 440–453, Berlin, Heidelberg. Springer-Verlag.

Biehl, M. (2010). Literature Study on Design Rationale and Design Decision Documentation for Architecture Descriptions. Technical report, Embedded Control Systems, Royal Institute of Technology. Version 20101101204500 TRITA-MMK 2010:06.

Bjørnson, F. O. and Dingsøyr, T. (2008). Knowledge management in software engineering: A systematic review of studied concepts, findings and research methods used. **Information and Software Technology**, 50(11):1055 – 1068.

Blom, H., Lönn, H., Hagl, F., Papadopoulos, Y., Reiser, M.-O., Sjöstedt, C.-J., Chen, D.-J., and Kolagari, R. T. (2012). EAST-ADL – An Architecture Description Language for Automotive Software-Intensive Systems. White Paper Version M2.1.10.

Bluetooth SIG (2013). Specification of the bluetooth®system, specification volume 1, architecture & terminology overview. Covered Core Package version: 4.1 - Publication date: 03 December 2013.

Boehm, B. (2002). Get ready for agile methods, with care. **Computer**, 35(1):64–69.

Boehm, B. W. (1986). A spiral model of software development and enhancement. **SIGSOFT Softw. Eng. Notes**, 11(4):14–24.

Böhm, C. and Jacopini, G. (1966). Flow diagrams, turing machines and languages with only two formation rules. **Commun. ACM**, 9(5):366–371.

Booch, G. (1991). **Object-oriented Analysis and Design with Applications**. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA.

Bosch, J. (2000). **Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach**. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Bosch, J. (2004). Software architecture: The next step. In **Proc. of the 1st European Workshop on Software Architecture**, pages 194–199. Springer.

Bracewell, R., Wallace, K., Moss, M., and Knott, D. (2009). Capturing design rationale. **Computer-Aided Design**, 41(3):173 – 186.

Braden, R. (1989). Requirements for Internet Hosts - Communication Layers. RFC 1122.

Brooks, F.P., J. (1987). No silver bullet - essence and accidents of software engineering. **Computer**, 20(4):10–19.

Broy, M. and Denert, E., editors (2002). **Software Pioneers - Contributions to Software Engineering**. Springer.

Buchmann, T., Dotor, A., and Westfechtel, B. (2009). Triple graph grammars or triple graph transformation systems? In Chaudron, M. R., editor, **Models in Software Engineering**, pages 138–150. Springer-Verlag, Berlin, Heidelberg.

Burge, J. E. (2005). **Software Engineering Using design RATionale**. PhD thesis, Worcester Polytechnic Institute.

Burge, J. E. and Brown, D. C. (2008). Software engineering using RATionale. **Journal of Systems and Software**, 81(3):395 – 413.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). **Pattern-oriented Software Architecture: A System of Patterns**. John Wiley & Sons, Inc., New York, NY, USA.

Bézivin, J. (2005). On the unification power of models. **Software & Systems Modeling**, 4(2):171–188.

Cailliau, A. and van Lamsweerde, A. (2012). A probabilistic framework for goal-oriented risk analysis. In **Requirements Engineering Conference (RE), 2012 20th IEEE International**, pages 201–210.

Cailliau, A. and van Lamsweerde, A. (2014). Integrating exception handling in goal models. In **Requirements Engineering Conference (RE), 2014 IEEE 22nd International**, pages 43–52.

Campbell, D. T. and Stanley, J. C. (1966). **Experimental and Quasi-Experimental Designs for Research**. Houghton Mifflin Company. Library of Congress Catalogue Card Number 81-80806, reprinted from Handbook of Research on Teaching (1963) by Houghton Mifflin Company.

Carrillo de Gea, J. M., Joaquín, N., Fernández Alemán, J. L., Toval, A., Ebert, C., and Vizcaíno, A. (2012). Requirements engineering tools: Capabilities, survey and assessment. **Information and Software Technology**, 54(10):1142 – 1157.

Carroll, J. M., editor (1996). **Design Rationale: Concepts, Techniques, and Use**. L. Erlbaum Associates Inc., Hillsdale, NJ, USA.

Carver, J. C., Jaccheri, L., Morasca, S., and Shull, F. (2010). A checklist for integrating student empirical studies with research and teaching goals. **Empirical Software Engineering**, 15(1):35–59.

Chen, L., Ali Babar, M., and Nuseibeh, B. (2013). Characterizing architecturally significant requirements. **Software, IEEE**, 30(2):38–45.

Cicchetti, A., Ruscio, D. D., Eramo, R., and Pierantonio, A. (2010). JTL: A bidirectional and change propagating transformation language. In Malloy, B. A., Staab, S., and van den Brand, M., editors, **proc. of the 3rd Int. Conf on Software Language Engineering**, volume 6563 of **LNCS**, pages 183–202. Springer.

Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., and Little, R. (2002). **Documenting Software Architectures: Views and Beyond**. Pearson Education.

Clements, P. C. and Bass, L. (2010). Relating business goals to architecturally significant requirements for software systems. Technical Report CMU/SEI-2010-TN-018, SEI, Carnegie Mellon University.

Coad, P. and Yourdon, E. (1991). **Object-oriented Design**. Yourdon Press, Upper Saddle River, NJ, USA.

Cohn, M. and Ford, D. (2003). Introducing an agile process to an organization [software development]. **Computer**, 36(6):74–78.

Cook, T. D. and Campbell, D. T. (1979). **Quasi-Experimentation: Design & Analysis Issues for Field Settings**. Houghton Mifflin Company.

Coplien, J. O. (1994). A development process generative pattern language. In **Proc. of the 1st Pattern Languages of Programs Conference (PLoP/94)**.

Curtis, B., Krasner, H., and Iscoe, N. (1988). A field study of the software design process for large systems. **Commun. ACM**, 31(11):1268–1287.

Curtis, W., Krasner, H., Shen, V., and Iscoe, N. (1987). On building software process models under the lamppost. In **Proc. of the 9th Int'l Conference on Software Engineering**, ICSE '87, pages 96–103, Los Alamitos, CA, USA. IEEE Computer Society Press.

Czarnecki, K. and Helsen, S. (2003). Classification of model transformation approaches. In **OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture**.

Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. **IBM Systems Journal**, 45(3):621–645.

Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R., editors (1972). **Structured Programming**. Academic Press Ltd., London, UK, UK.

Dahl, O.-J., Myrhraug, B., and Nygaard, K. (1970). **SIMULA Common Base Language**. Norwegian Computing Center, revised edition s-22 edition.

Daigneau, R. (2011). **Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services**. Addison-Wesley Professional.

Dajsuren, Y., van den Brand, M., Serebrenik, A., and Huisman, R. (2012). Automotive adls: A study on enforcing consistency through multiple architectural levels. In Grassi, V., Mirandola, R., Buhnova, B., and Vallecillo, A., editors, **Proc. of the 8th International ACM Sigsoft Conference on the Quality of Software Architectures (QoSA)**, pages 71–80. ACM.

Dankova, P. (2009). Main aspects of enterprise architecture concept. **Economic Alternatives**, pages 102–114.

Dardenne, A., van Lamsweerde, A., and Fickas, S. (1993). Goal-directed requirements acquisition. In **Science of Computer Programming**, volume 20, pages 3–50, North Holland.

Darimont, R., Delor, E., Massonet, P., and van Lamsweerde, A. (1997). Grail/kaos: An environment for goal-driven requirements engineering. In **Software Engineering, 1997., Proceedings of the 1997 (19th) International Conference on**, pages 612–613.

Dashofy, E., van der Hoek, A., and Taylor, R. (2001). A highly-extensible, xml-based architecture description language. In **Proc. of the Working IEEE/IFIP Conference on Software Architecture**, pages 103–112.

Dashofy, E. M. (2007). **Supporting Stakeholder-Driven, Multi-View Software Architecture Modeling**. PhD thesis, University of California, Irvine, Irvine, CA.

Dashofy, E. M., Hoek, A. v. d., and Taylor, R. N. (2005). A comprehensive approach for the development of modular software architecture description languages. **ACM Trans. Softw. Eng. Methodol.**, 14(2):199–245.

Dashofy, E. M., van der Hoek, A., and Taylor, R. N. (2002). An infrastructure for the rapid development of xml-based architecture description languages. In **Proceedings of the 24th International Conference on Software Engineering**, ICSE '02, pages 266–276, New York, NY, USA. ACM.

de Boer, R., Farenhorst, R., van der Ven, J., Clerc, V., Deckers, R., Lago, P., and van Vliet, H. (2006). Structuring software architecture project memories. In **Proc. of the 8th Int'l Workshop on Learning Software Organizations (LSO'06)**.

de Boer, R. C., Farenhorst, R., Lago, P., Vliet, H., Clerc, V., and Jansen, A. (2007). Architectural knowledge: Getting to the core. In Overhage, S., Szyperski, C. A., Reussner, R., and Stafford, J. A., editors, **Software Architectures, Components, and Applications**, volume 4880 of **Lecture Notes in Computer Science**, pages 197–214. Springer Berlin Heidelberg.

Delatte, B., Heitz, M., and Muller, J. (1992). **Hood reference manual 3.1**. Prentice Hall. Hood technical group.

Deng, G., Balasubramanian, J., Otte, W., Schmidt, D., and Gokhale, A. (2005). Dance: A qos-enabled component deployment and configuration engine. In Dearle, A. and Eisenbach, S., editors, **Component Deployment**, volume 3798 of **Lecture Notes in Computer Science**, pages 67–82. Springer Berlin Heidelberg.

Détienne, F. (1990). Expert Programming Knowledge: a Schema-Based Approach. In Hoc, J.-M., Green, T., Samurcay, R., and Gilmore, D., editors, **Psychology of Programming**, People and Computer series, pages 205–222. Academic Press.

EAST-ADL, A. (2013). EAST-ADL Domain Model Specification, version V 2.1.12.

Edmonds, E. A. (1974). A process for the development of software for non-technical users as an adaptive system. **General Systems**, XIX:215–218.

Ehrig, H., Pfender, M., and Schneider, H. J. (1973). Graph-grammars: An algebraic approach. In **IEEE Conference Record of 14th Annual Symposium on Switching and Automata Theory**, pages 167–180.

Ehrig, K., Guerra, E., Lara, J. D., Lengyel, L., Prange, U., Taentzer, G., Varro, D., and Varro-Gyapay, S. (2005). Model transformation by graph transformation: A comparative study. In **Proc. of Model Transformation in Practice Workshop**, MoDELS '05, pages 71–80.

Englebert, V. and Vermaut, F. (2004). Attribute-based refinement of software architectures. In **4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04)**, page 301, Los Alamitos, CA, USA. IEEE Computer Society.

Fahmy, H. and Blostein, D. (1992). A survey of graph grammars: theory and applications. In **Pattern Recognition, 1992. Vol.II. Conference B: Pattern Recognition Methodology and Systems, Proceedings., 11th IAPR International Conference on**, pages 294–298.

Farenhorst, R. and de Boer, R. C. (2009). Knowledge management in software architecture: State of the art. In Ali Babar, M., Dingsøyr, T., Lago, P., and van Vliet, H., editors, **Software Architecture Knowledge Management**, pages 21–38. Springer Berlin Heidelberg.

Feiler, P. H., Gluch, D. P., and Woodham, K. (2010). Case Study: Model-Based Analysis of the Mission Data System Reference Architecture. Technical Report CMU/SEI-2010-TR-003, SEI, Carnegie Mellon University.

Feiler, P. H., Seibel, J., and Wrage, L. (2012). What's New in V2 of the Architecture Analysis & Design Language Standard? Technical Report CMU/SEI-2011-SR-011, SEI, Carnegie Mellon University.

Feller, P. H., Gluch, D. P., and Hudack, J. J. (2006). The Architecture Analysis & Design Language (AADL) : An Introduction. Technical Report CMU/SEI-2006-TN-011, SEI, Carnegie Mellon University.

Fielding, R. T. (2000). **Architectural Styles and the Design of Network-based Software Architectures**. PhD thesis, University of California, Irvine. AAI9980887.

Floyd, R. W. (1967). Assigning meanings to programs. In **Proc. of Symposia in Applied Mathematics**, volume 19, pages 19–32. American mathematical Society.

Fowler, M. (2002). **Patterns of Enterprise Application Architecture**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Fowler, M. (2010). **Domain Specific Languages**. The Addison-Wesley Signature Series. Addison-Wesley Professional, 1st edition.

France, R., Ghosh, S., Dinh-Trong, T., and Solberg, A. (2006). Model-driven development using uml 2.0: promises and pitfalls. **Computer**, 39(2):59–66.

France, R. and Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. In **2007 Future of Software Engineering**, FOSE '07, pages 37–54, Washington, DC, USA. IEEE Computer Society.

Fuentes, L. and Gámez, N. (2007). Adding aspects to xadl 2.0 for software product line architectures. In Pohl, K., Heymans, P., Kang, K. C., and Metzger, A., editors, **Proc. of the Internation Workshop on Variability Modeling of Software Intensive Systems (VAMOS'07)**, volume 2007-01 of **Lero Technical Report**, pages 87–96.

Gacek, C., Abd-allah, A., Clark, B., and Boehm, B. (1995). On the definition of software system architecture. In **Proc. of ICSE 17 Software Architecture Workshop**.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). **Design Patterns: Elements of Reusable Object-oriented Software**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. M. (1993). Design patterns: Abstraction and reuse of object-oriented design. In **Proc. of the 7th European Conf. on Object-Oriented Programming (ECOOP'93)**, pages 406–431.

Gardner, T., Griffin, C., Koehler, J., and Hauser, R. (2003). A review of omg mof 2.0 query/views/transformations submissions and recommendations towards the final standard. In **MetaModelling for MDA Workshop**, pages 178–197.

Garlan, D., Monroe, R. T., and Wile, D. (1997). Acme: An architecture description interchange language. In **Conference of the Centre for Advanced Studies on Collaborative research (CASCON 97)**, pages 169–183, Toronto, Ontario.

Garlan, D., Monroe, R. T., and Wile, D. (2000). Acme: Architectural description of component-based systems. In Leavens, G. T. and Sitaraman, M., editors, **Foundations of Component-Based Systems**, pages 47–68. Cambridge University Press.

Garlan, D. and Shaw, M. (1994). An introduction to software architecture. Technical Report CMU-CS-94-166, School of Computer Science, Carnegie Mellon University.

Giese, H. and Wagner, R. (2006). Incremental model synchronization with triple graph grammars. In **Proc. of the 9th Int'l Conf. on Model Driven Engineering Languages and Systems**, MoDELS'06, pages 543–557, Berlin, Heidelberg. Springer-Verlag.

Gilb, T. (1977). **Software metrics**. Winthrop Publishers.

Gilb, T. (1981). Evolutionary development. **SIGSOFT Softw. Eng. Notes**, 6(2):17–17.

Gilb, T. (1988). **Principles of Software Engineering Management**. Addison-Wesley, Boston, MA, USA.

Gilberth, F. B. and Gilberth, L. M. (1921). Process charts. In **Annual Meeting of the American Society of Mechanical Engineers**.

Gilson, F. (2010). A transformational approach for component-based distributed architectures. In **Preliminary Proc. of the MODELS 2010 Doctoral Symposium**, pages 25–30.

Gilson, F. and Englebert, V. (2011a). Rationale, decisions and alternatives traceability for architecture design. In **Proc. of the 5th ECSA Workshop on Traceability, Dependencies and Software Architecture (TDSA'11)**. ACM.

Gilson, F. and Englebert, V. (2011b). Towards handling architecture design, variability and evolution with model transformations. In **Proc. of the 5th Workshop on Variability Modeling of Software-Intensive Systems**, pages 39–48. ACM.

Gilson, F. and Englebert, V. (2014a). A domain specific language for stepwise design of software architectures. In **Proc. of the 2nd Int'l Conf. on Model-Driven Engineering and Software Development**, pages 67–78. SciTePress.

Gilson, F. and Englebert, V. (2014b). Transformation-wise design of software architectures. In **To appear in Communications in Computer and Information Science**. Springer.

Gilson, F. and Englebert, V. (2015). Software architecture design by stepwise model transformations : a comparative case study. In **Proc. of the 3rd Int'l Conf. on Model-Driven Engineering and Software Development**, pages 134–145. SciTePress.

Gilson, F., Englebert, V., and Matulevičius, R. (2008). A large scope transformational approach for distributed architecture design. In **Proc. of the 2nd Eur. Conf. on Software Architecture**, volume 5292/2008 of **Lecture Notes in Computer Science**, pages 330–333. Springer.

Goldstine, H. H. and von Neumann, J. (1947). **Planning and Coding of Problems for an Electronic Computing Instrument**, chapter 7.0 General Principles of Coding and Flow-Diagramming, pages 1 – 23. Institute for Advanced Study, Princeton, New Jersey.

Gottmann, S., Hermann, F., Nachtigall, N., Braatz, B., Ermel, C., Ehrig, H., and Engel, T. (2013). Correctness and completeness of generalised concurrent model synchronisation based on triple graph grammars. In Baudry, B., Dingel, J., Lucio, L., and Vangheluwe, H., editors, **Proceedings of the 2nd Workshop on the Analysis of Model Transformations (AMT 2013)**, volume 1077 of **CEUR Workshop Proceedings**.

Grenning, J. (2002). Planning poker or how to avoid analysis paralysis while release planning. available online.

Gross, D. and Yu, E. (2001). Evolving system architecture to meet changing business goals: an agent and goal-oriented approach. In **From Software Requirements to Architectures (STRAW ICSE'01 Workshop)**, pages 13–21, Toronto, Canada.

Gruber, T., Baudin, C., Boose, J., and Weber, J. (1991). Design rationale capture as knowledge acquisition tradeoffs in the design of interactive tools. In Birnbaum, L. and Collins, G., editors, **Proc. of the 8th International Workshop In Machine Learning**, pages 3–12. Morgan Kaufman.

Guerra, E., de Lara, J., Kolovos, D. S., Paige, R. F., and dos Santos, O. M. (2010). transml: A family of languages to model model transformations. In **Proc. of the 13th Int'l Conf. on Model Driven Engineering Languages and Systems: Part I**, MODELS'10, pages 106–120, Berlin, Heidelberg. Springer-Verlag.

Guerra, E., Lara, J., Kolovos, D. S., Paige, R. F., and Santos, O. (2013). Engineering model transformations with transml. **Software & Systems Modeling**, 12(3):555–577.

Hansen, M. T., Nohria, N., and Tierney, T. (1999). What's your strategy for managing knowledge. **Harvard Business Review**, 77(2):106 – 116.

Harrison, N., Avgeriou, P., and Zdun, U. (2007). Using patterns to capture architectural decisions. **Software, IEEE**, 24(4):38–45.

Heer, J. and Agrawala, M. (2006). Software design patterns for information visualization. **IEEE Transactions on Visualization and Computer Graphics**, 12(5):853–860.

Hermann, F., Ehrig, H., Ermel, C., and Orejas, F. (2012). Concurrent model synchronization with conflict resolution based on triple graph grammars. In Lara, J. and Zisman, A., editors, **Fundamental Approaches to Software Engineering**, volume 7212 of **Lecture Notes in Computer Science**, pages 178–193. Springer Berlin Heidelberg.

Hildebrandt, S., Lambers, L., Giese, H., Rieke, J., Greenyer, J., Schäfer, W., Anjorin, A., and Schürr, A. (2013). A survey of triple graph grammar tools. In **Proc. of the 2nd Int'l Workshop on Bidirectional Transformations (BX 2013**, volume 57 of **Electronic Communications of the EASST**.

Hilliard, R. and Rice, T. (1998). Expressiveness in architecture description languages. In **Proceedings of the Third International Workshop on Software Architecture**, ISAW '98, pages 65–68, New York, NY, USA. ACM.

Hoare, C. (1972). Proof of correctness of data representations. **Acta Informatica**, 1(4):271–281.

Hofmeister, C., Kruchten, P., Nord, R. L., Obbink, H., Ran, A., and America, P. (2007). A general model of software architecture design derived from five industrial approaches. **J. Sys. Softw.**, 80(1):106–126.

Hofmeister, C., Nord, R., and Soni, D. (1999). **Applied Software Architecture**. Addison-Wesley, Boston, USA.

Höst, M., Regnell, B., and Wohlin, C. (2000). Using students as subjects – a comparative study of students and professionals in lead-time impact assessment. **Empir. Soft. Eng.**, 5(3):201–214.

IEEE (2000). Recommended practices for architectural description of software-intensive systems. **IEEE Std 1471-2000**.

IEEE (2013). 802.3-2012 - ieee standard for ethernet. IEEE Computer Society Standard. WG802.3 - Ethernet Working Group.

International Organization for Standardization (1994). Information technology – Open Systems Interconnection – Basic Reference Model: The Basic Model. ISO/IEC 7498-1:1994.

ISO/IEC/IEEE (2011). Systems and software engineering – architecture description. ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000).

Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. (1992). **Object-oriented software engineering - a use case driven approach**. Addison-Wesley.

Jansen, A. (2008). **Architectural Design Decisions**. PhD thesis, Institute of Mathematics and Computing Science, University of Groningen.

Jansen, A. and Bosch, J. (2005). Software architecture as a set of architectural design decisions. In **Proc. of the 5th Working Conf. on Software Architecture**, pages 109–120, Washington, DC, USA. IEEE Computer Society.

Jansen, A., van der Ven, J., Avgeriou, P., and Hammer, D. K. (2007). Tool support for architectural decisions. In **Proc. of the 6th Working IEEE/IFIP Conf. on Software Architecture**, pages 4–, Washington, DC, USA. IEEE Computer Society.

Jarczyk, A., Loffler, P., and Shipman, F. (1992). Design rationale for software engineering: a survey. In **System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on**, volume ii, pages 577–586 vol.2.

Jensen, K. (1996). An introduction to the practical use of coloured petri nets. In **Petri Nets (2)**, pages 237–292.

Jones, S. (1983). Stereotypy in pictograms of abstract concepts. **Ergonomics**, 26(6):605–611.

Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). Atl: A model transformation tool. **Science of Computer Programming**, 72(1–2):31 – 39. Special Issue on Second issue of experimental software and toolkits (EST).

Jouault, F. and Bézivin, J. (2006). Km3: A dsl for metamodel specification. In **Proc. of the 8th IFIP Formal Methods for Open Object-Based Distributed Systems**, volume 4037 of **Lecture Notes in Computer Science**, pages 171–185. Springer.

Jouault, F. and Kurtev, I. (2005). Transforming models with ATL. In **Model Transformations in Practice (MTIP) Workshop at ACM/IEEE 8th Int'l Conference on Model Driven Engineering Languages and Systems**.

Jouault, F. and Tisi, M. (2010). Towards incremental execution of atl transformations. In Tratt, L. and Gogolla, M., editors, **Theory and Practice of Model Transformations**, volume 6142 of **Lecture Notes in Computer Science**, pages 123–137. Springer Berlin Heidelberg.

Kalnins, A., Celms, E., and Sostaks, A. (2005). Model transformation approach based on mola. In **Model Transformations in Practice (MTIP) Workshop at ACM/IEEE 8th Int. Conf. on Model Driven Engineering Languages and Systems**. Springer.

Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., and Wimmer, M. (2012). Model transformation by-example: A survey of the first wave. In Düsterhöft, A., Klettke, M., and Schewe, K.-D., editors, **Conceptual Modelling and Its Theoretical Foundations**, volume 7260 of **Lecture Notes in Computer Science**, pages 197–215, Berlin, Heidelberg. Springer-Verlag.

Kazman, R., Klein, M., and Clements, P. (2000). ATAM: method for architecture evaluation. Technical Report CMU/SEI-2000-TR-004, SEI, Carnegie Mellon University.

Kelly, S. and Tolvanen, J.-P. (2008). **Domain-Specific Modeling: Enabling Full Code Generation**. John Wiley & Sons, Inc.

Kent, S. (2002). Model driven engineering. In Butler, M., Petre, L., and Sere, K., editors, **Integrated Formal Methods**, volume 2335 of **Lecture Notes in Computer Science**, pages 286–298. Springer Berlin Heidelberg.

Kleppe, A. G., Warmer, J., and Bast, W. (2003). **MDA Explained: The Model Driven Architecture: Practice and Promise**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Königs, A. and Schürr, A. (2006). Tool integration with triple graph grammars - a survey. **Electronic Notes in Theoretical Computer Science**, 148(1):113 – 150. Proc. of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004).

Kroll, P. and Kruchten, P. (2003). **The Rational Unified Process Made Easy: a practionner's guide to the RUP**. Addison Wesley Pearson Education, Inc.

Krosnick, J. A. and Presser, S. (2010). Question and questionnaire design. In Marsdenand, P. V. and Wright, J. D., editors, **Handbook of Survey Research, Second Edition**, pages 263–313. Emerald Group Publishing Limited.

Kruchten, P. (1995). The 4+1 view model of architecture. **IEEE Softw.**, 12(6):42–50.

Kruchten, P. (2003). **The Rational Unified Process: An Introduction**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition.

Kruchten, P., Lago, P., and Vliet, H. (2006). Building up and reasoning about architectural knowledge. In Hofmeister, C., Crnkovic, I., and Reussner, R., editors, **Quality of Software Architectures**, volume 4214 of **Lecture Notes in Computer Science**, pages 43–58. Springer Berlin Heidelberg.

Krutchen, P. (2004). An ontology of architectural design decisions in software intensive systems. In **Proc. of the 2nd Groningen Workshop on Software Variability Management**, pages 54–61.

Kunz, W. and Rittel, H. (1970). Issues as elements of information systems. Working Paper 131, Institute of Urban and Regional Development, University of California, Berkeley, California.

Kusel, A., Etzlstorfer, J., Kapsammer, E., Langer, P., Retschitzegger, W., Schönböck, J., Schwinger, W., and Wimmer, M. (2013a). A survey on incremental model transformation approaches. In **Proc of Models and Evolution Workshop @ MoDELS**.

Kusel, A., Schönböck, J., Wimmer, M., Kappel, G., Retschitzegger, W., and Schwinger, W. (2013b). Reuse in model-to-model transformation languages: are we there yet? **Software & Systems Modeling**, pages 1–36.

Lago, P., Farenhorst, R., Avgeriou, P., Boer, R. C., Clerc, V., Jansen, A., and Vliet, H. (2010). The griffin collaborative virtual community for architectural knowledge management. In Mistrík, I., Grundy, J., Hoek, A., and Whitehead, J., editors, **Collaborative Software Engineering**, pages 195–217. Springer Berlin Heidelberg.

Lai, Q. and Carpenter, A. (2012). Defining and verifying behaviour of domain specific language with fuml. In **Proceedings of the Fourth Workshop on Behaviour Modelling - Foundations and Applications**, BM-FA '12, pages 1:1–1:7, New York, NY, USA. ACM.

Langer, P., Wimmer, M., and Kappel, G. (2010). Model-to-model transformations by demonstration. In Tratt, L. and Gogolla, M., editors, **Theory and Practice of Model Transformations**, volume 6142 of **Lecture Notes in Computer Science**, pages 153–167. Springer Berlin Heidelberg.

Lapkin, A., Allega, P., Burke, B., Burton, B., Bittler, R. S., Handler, R. A., James, G. A., Robertson, B., Newman, D., Weiss, D., Buchanan, R., and Gall, N. (2008). Gartner Clarifies the Definition of the Term 'Enterprise Architecture'. ID Number: G00156559.

Larman, C. (2003). **Agile and Iterative Development: A Manager's Guide**. Addison Wesley.

Larman, C. and Basili, V. R. (2003). Iterative and incremental developments. a brief history. **Computer**, 36(6):47–56.

Leblebici, E., Anjorin, A., Schürr, A., Hildebrandt, S., Rieke, J., , and Greenyer, J. (2014). A comparison of incremental triple graph grammar tools. In **Proc. of the 13th Int'l Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT'14)**, volume X of **Electronic Communications of the EASST**.

Lee, J. (1989). Decision representation language (DRL) and its support environment. Technical report, MIT Artificial Intelligence Laboratory.

Lee, J. (1991). Extending the potts and bruns model for recording design rationale. In **Proc. of the Int'l Conf. on Software Engineering**, pages 114–125.

Leff, A. and Rayfield, J. (2010). Eds: An elastic data-service for situational applications. In **2010 IEEE Int'l Conf. on Web Services (ICWS'10)**, pages 187–194.

Lehman, M. M. and Belady, L. A., editors (1985). **Program Evolution: Processes of Software Change**. Academic Press Professional, Inc., San Diego, CA, USA.

Liu, L., Li, T., and Peng, F. (2010). Why requirements engineering fails: A survey report from china. In **Requirements Engineering Conference (RE), 2010 18th IEEE International**, pages 317–322.

Liu, L. and Yu, E. (2004). Designing information systems in social context: A goal and scenario modelling approach. **Information Systems**, 29(2):187–203.

Luckham, D. C., Kenney, J. J., Augustin, L. M., Vera, J., Bryan, D., and Mann, W. (1995). Specification and analysis of system architecture using rapide. **IEEE Trans. Softw. Eng.**, 21(4):336–355.

Magee, J. (1999). Behavioral analysis of software architectures using ltsa. In **Proc. of the 21st Int. Conf. on Software engineering (ICSE 99)**, pages 634–637, New York, NY, USA. ACM.

Magee, J., Dulay, N., Eisenbach, S., and Kramer, J. (1995). Specifying distributed software architectures. In Schafer, W. and Botella, P., editors, **Proc of the 5th European Software Engineering Conf. (ESEC'95)**, pages 137–153, Sitges, Spain. Springer.

Malavolta, I., Lago, P., Muccini, H., Pelliccione, P., and Tang, A. (2013). What industry needs from architectural languages: A survey. **IEEE Trans. Softw. Eng.**, 39(6):869–891.

Marca, D. and McGowan, C. (1982). Static and dynamic data modeling for information system design. In **Proceedings of the 6th International Conference on Software Engineering**, ICSE '82, pages 137–146, Los Alamitos, CA, USA. IEEE Computer Society Press.

Marosin, D., Ghanavati, S., and van der Linden, D. (2014). A principle-based goal-oriented requirements language (GRL) for enterprise architecture. In Dalpiaz, F. and Horkoff, J., editors, **Proc. of the 7th Int'l i* Workshop co-located with the 26th Int'l Conf. on Advanced Information Systems Engineering (CAiSE 2014)**, volume 1157 of **CEUR Workshop Proceedings**. CEUR-WS.org.

Mavin, A. (2012). Listen, then use ears. **Software, IEEE**, 29(2):17–18.

Mavin, A. and Wilkinson, P. (2010). Big ears (the return of "easy approach to requirements engineering"). In **Proc. of the 18th IEEE Int. Requirements Engineering Conf.**, pages 277–282.

Mayerhofer, T., Langer, P., Wimmer, M., and Kappel, G. (2013). xmof: Executable dsmls based on fuml. In Erwig, M., Paige, R., and Wyk, E., editors, **Software Language Engineering**, volume 8225 of **Lecture Notes in Computer Science**, pages 56–75. Springer International Publishing.

McCracken, D. D. and Jackson, M. A. (1982). Life cycle concept considered harmful. **SIGSOFT Softw. Eng. Notes**, 7(2):29–32.

Medvidovic, N., Oreizy, P., Robbins, J. E., and Taylor, R. N. (1996). Using object-oriented typing to support architectural design in the c2 style. In **Proc. of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering**, SIGSOFT '96, pages 24–32, New York, NY, USA. ACM.

Medvidovic, N., Rosenblum, D. S., and Taylor, R. N. (1999). A language and environment for architecture-based software development and evolution. In **Proceedings of the 21st International Conference on Software Engineering**, ICSE '99, pages 44–53, New York, NY, USA. ACM.

Medvidovic, N. and Taylor, R. N. (2000). A Classification and Comparison Framework for Software Architecture Description Languages. **IEEE Trans. Softw. Eng.**, 26(1):70–93.

Mens, T., Czarnecki, K., and Gorp, P. V. (2005). A taxonomy of model transformations. In Bezivin, J. and Heckel, R., editors, **Language Engineering for Model-Driven Software Development**, number 04101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI).

Mentz, J., Kotze, P., and van der Merwe, A. (2012). A comparison of practitioner and researcher definitions of enterprise architecture using an interpretation method. In Möller, C. and Chaudhry, S., editors, **Advances in Enterprise Information Systems II**, pages 11–26. CRC Press.

Metzger, A. (2005). A systematic look at model transformations. In Beydeda, S., Book, M., and Gruhn, V., editors, **Model-Driven Software Development**, pages 19–33. Springer Berlin Heidelberg.

Meyer, B. (1992). Applying "design by contract". **Computer**, 25(10):40–51.

Milner, R. (1982). **A Calculus of Communicating Systems**. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Minsky, M. (1974). A framework for representing knowledge. Technical Report Memo 306, Massachussets Institute of Technology - Artificial Intelligence Laboratory. also appear in "The Psychology of Computer Vision, P. Winston (Ed.), McGraw-Hill, 1975".

Monroe, R. T. (2001). Capturing software architecture design expertise with Armani. Technical Report CMU-CS-98-163, School of Computer Science, Carnegie Mellon University. originally published October, 1998 (version 2.3).

Morrison, R., Kirby, G., Balasubramaniam, D., Mickan, K., Oquendo, F., Cimpan, S., Warboys, B., Snowdon, B., and Greenwood, R. (2004). Support for evolving software architectures in the archware adl. In **Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on**, pages 69–78.

Muller, P.-A., Fleurey, F., and Jézéquel, J.-M. (2005a). Weaving executability into object-oriented meta-languages. In **Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems**, MoDELS'05, pages 264–278, Berlin, Heidelberg. Springer-Verlag.

Muller, P.-A., Fleurey, F., Vojtisek, D., Drey, Z., Pollet, D., Fondement, F., Studer, P., and Jézéquel, J.-M. (2005b). On Executable Meta-Languages applied to Model Transformations. In **Model Transformations In Practice Workshop**, Montego Bay, Jamaica.

Mylopoulos, J., Chung, L., and Yu, E. (1999). From object-oriented to goal-oriented requirements analysis. **Commun. ACM**, 42(1):31–37.

Nagl, M. (1978). A tutorial and bibliographical survey on graph grammars. In Claus, V., Ehrig, H., and Rozenberg, G., editors, **Graph-Grammars and Their Application to Computer Science and Biology**, volume 73 of **Lecture Notes in Computer Science**, pages 70–126. Springer.

Nagl, M. (1987). Set theoretic approaches to graph grammars. In Ehrig, H., Nagl, M., Rozenberg, G., and Rosenfeld, A., editors, **Graph-Grammars and Their Application to Computer Science**, volume 291 of **Lecture Notes in Computer Science**, pages 41–54. Springer Berlin Heidelberg.

Naslavsky, L., Xu, L., Dias, M., Ziv, H., and Richardson, D. J. (2004). Extending xadl with statechart behavioral specification. In **Proc. of the Twin Workshops on Architecting Dependable Systems at International Conference of Software Engineering**, pages 22–26.

Naur, P. and Randell, B., editors (1969). **Software Engineering - Report on a conference sponsored by the NATO Science Comittee**.

Obbink, H., Muller, J., America, P., van Ommering, R., Muller, G., van der Sterren, W., and Wijnstra, J. (2000). COPA: a component-oriented platform architecting method for families of software-intensive electronic products (tutorial). In **Proc. of the First Software Product Line Conference (SPLC)**.

Object Management Group (1997). UML Specification version 1.1. OMG document ad/97-08-11.

Object Management Group (2001). OMG Unified Modeling Language Specification, version 1.4. OMG document formal/2001-09-67.

Object Management Group (2002). Request for Proposal: MOF 2.0 Query / Views / Transformations RFP. OMG document ad/2002-04-10.

Object Management Group (2003). Common Warehouse Metamodel (CWM) Specification, version 1.1, Volume 1. OMG document formal/03-03-02.

Object Management Group (2006). Deployment and Configuration of Component-based Distributed Applications, version 4.0. OMG specification formal/06-04-02.

Object Management Group (2007). OMG Systems Modeling Language (OMG SysML®), V1.0. OMG document formal/2007-09-01.

Object Management Group (2008). Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification - Version 1.0. OMG document formal/2008-04-03.

Object Management Group (2009). Production Rule Representation (PRR), version 1.0. OMG document formal/2009-12-01.

Object Management Group (2011a). Business process model and notation (bpmn), version 2.0. OMG document formal/2011-01-03.

Object Management Group (2011b). Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification - Version 1.1. OMG document formal/2011-01-01.

Object Management Group (2011c). OMG Unified Modeling Language (OMG UML), Infrastructure, version 2.4.1. OMG document formal/2011-08-05.

Object Management Group (2011d). OMG Unified Modeling Language (OMG UML), Superstructure, version 2.4.1. OMG document formal/2011-08-06.

Object Management Group (2011e). UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, version 1.1. OMG document formal/2011-06-02.

Object Management Group (2012a). Common Object Request Broker Architecture (CORBA) Specification, version 3.3 - Part 1: CORBA Interfaces. OMG document formal/2012-11-12.

Object Management Group (2012b). Common Object Request Broker Architecture (CORBA) Specification, version 3.3 - Part 2: CORBA Component Model. OMG document formal/2012-11-16.

Object Management Group (2012c). OMG Object Constraint Language (OCL), Version 2.3.1. OMG document formal/2012-01-01.

Object Management Group (2012d). OMG Systems Modeling Language (OMG SysML™), version 1.3. OMG document formal/2012-06-01.

Object Management Group (2013a). Action Language for Foundational UML (Alf) - Concrete Syntax for a UML Action Language, version 1.0.1. OMG document formal/2013-09-01.

Object Management Group (2013b). OMG MOF 2 XMI Mapping Specification, version 2.4.1. OMG document formal/2013-06-03.

Object Management Group (2013c). Semantics of a Foundational Subset for Executable UML Models (fUML), version 1.1. OMG document formal/2013-08-06.

Object Management Group (2014a). Decision Model and Notation - Beta1. OMG document dtc/2014-02-01.

Object Management Group (2014b). Meta Object Facility (MOF) Core Specification - Version 2.4.2. OMG document formal/2014-04-03.

Object Management Group Architecture Board ORMSC (2001). Model Driven Architecture. OMG document ormsc/2001-07-01.

Ono, K., Koyanagi, T., Abe, M., and Hori, M. (2002). Xslt stylesheet generation by example with wysiwyg editing. In **Proc. of the Symposium on Applications and the Internet**, SAINT '02, pages 150–161, Washington, DC, USA. IEEE Computer Society.

Open Group (2011). Open Group Standard TOGAF®Version 9.1. Document number: G116.

Open Group (2013). Open Group Standard ArchiMate®2.1 Specification. Document Number: C13L.

Oquendo, F. (2004). π-adl: an architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. **SIGSOFT Software Engineering Notes**, 29(3):1–14.

Oquendo, F. (2008a). Dynamic software architectures: Formally modelling structure and behaviour with π-adl. In **Proc of the Int'l Conf. on Software Engineering Advances (ICSEA'08)**, pages 352–359.

Oquendo, F. (2008b). Formal approach for the development of business processes in terms of service-oriented architectures using pi-adl. In **Service-Oriented System Engineering, 2008. SOSE '08. IEEE International Symposium on**, pages 154–159.

Oquendo, F., Warboys, B., Morrison, R., Dindeleux, R., Gallo, F., Garavel, H., and Occhipinti, C. (2004). Archware: Architecting evolvable software. In Oquendo, F., Warboys, B. C., and Morrison, R., editors, **Software Architecture**, volume 3047 of **Lecture Notes in Computer Science**, pages 257–271. Springer Berlin Heidelberg.

Pannok, P. and Vatanawood, W. (2013). An xadl extension for service oriented architecture design. In **Information Science and Applications (ICISA), 2013 International Conference on**, pages 1–3.

Parnas, D. L. and Clements, P. C. (1986). A rational design process: How and why to fake it. **IEEE Trans. Software Eng.**, 12:251–257.

Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. **SIGSOFT Software Engineering Notes**, 17(4):40–52.

Petre, M. (2013). Uml in practice. In **Proc. of the Int'l Conf. on Software Engineering**, ICSE'13, pages 722–731, Piscataway, NJ, USA. IEEE Press.

Pfaltz, J. L. and Rosenfeld, A. (1969). Web grammars. In **Proc. of th 1st Joint Conf. on Artificial Intelligence**, pages 609 – 619.

Pfleeger, S. L. (1995). Experimental design and analysis in software engineering. **Ann. Softw. Eng.**, 1:219–253.

Pinto, M., Fuentes, L., and Troya, J. M. (2003). Daop-adl: An architecture description language for dynamic component and aspect-based development. In Pfenning, F. and Smaragdakis, Y., editors, **Proc. of the 2nd Int. Conf. on Generative Programming and Component Engineering**, volume 2830 of **Lecture Notes in Computer Science**, pages 118–137. Springer.

Pratt, T. W. (1971). Pair grammars, graph languages and string-to-graph translations. **J. Comput. Syst. Sci.**, 5(6):560–595.

Prechelt, L., Unger-Lamprecht, B., Philippsen, M., and Tichy, W. (2002). Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. **Software Engineering, IEEE Transactions on**, 28(6):595–606.

Qayyum, Z. and Oquendo, F. (2008). The pi-adl.net project: An inclusive approach to adl compiler design. **W. Trans. on Comp.**, 7(5):414–423.

Ramesh, B. and Jarke, M. (2001). Toward reference models for requirements traceability. **IEEE Trans. Software Eng.**, 27(1):58 –93.

Ran, A. (2000). ARES conceptual framework for software architecture. In Jazayeri, M., Ran, A., and van der Linden, F., editors, **Software Architecture for Product Families: Principles and Practice**, pages 1–29. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Ráth, I., Bergmann, G., Ökrös, A., and Varró, D. (2008). Live model transformations driven by incremental pattern matching. In Vallecillo, A., Gray, J., and Pierantonio, A., editors, **Theory and Practice of Model Transformations**, volume 5063 of **Lecture Notes in Computer Science**, pages 107–121. Springer Berlin Heidelberg.

Rational Software (1998). Rational unified process: Best practices for software development teams. Technical Report White Paper, TP026B, Rev 11/01, Rational Software.

Rich, C. (1981). Inspection methods in programming. Technical Report AI-TR-604, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, MA, USA.

Rieke, J. and Sudmann, O. (2012). Specifying refinement relations in vertical model transformations. In Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., and Kolovos, D., editors, **Modelling Foundations and Applications**, volume 7349 of **Lecture Notes in Computer Science**, pages 210–225. Springer Berlin Heidelberg.

Robertson, J. and Robertson, S. (2012). Volere requirements specification template, edition 16. Atlantic Systems Guild.

Ross, D. (1977). Structured analysis (sa): A language for communicating ideas. **Software Engineering, IEEE Transactions on**, SE-3(1):16–34.

Royce, W. W. (1970). Managing the development of large software systems. In **Proc. of IEEE WESCON 26**, pages 1–9. IEEE Press.

Rumbaugh, J. R., Blaha, M. R., Lorensen, W., Eddy, F., and Premerlani, W. (1990). **Object-Oriented Modeling and Design**. Prentice Hall.

Rumelhart, D. (1980). Schemata: The building blocks of cognition. **Theoretical Issues in Reading Comprehension**, pages 33–58.

Rus, I. and Lindvall, M. (2002). Knowledge management in software engineering. **Software, IEEE**, 19(3):26–38.

Salvestrini, F., Carrozzo, G., Cruschelli, P., Chappel, A., Graham, J., Vrijders, S., Staessens, D., Tarzan, M., Bergesio, L., Trouva, E., Grasa, E., Vico, A., and Bermudo, C. (2013). D2.1 first phase use cases, requirements analysis, rina specifications and high-level software architecture. Investigating RINA as an Alternative to TCP/IP.

Scheer, A.-W. and Nüttgens, M. (2000). Aris architecture and reference models for business process management. In **Business Process Management, Models, Techniques, and Empirical Studies**, pages 376–389, London, UK, UK. Springer-Verlag.

Schmidt, D. (2006). Guest editor's introduction: Model-driven engineering. **Computer**, 39(2):25–31.

Schneider, H. J. (1971). Chomsky-like systems for partially ordered symbol sets. Technical report, Berlin : Technische Universität. (Informationsverarbeitung II). - Internal report.

Schürr, A. (1995). Specification of graph translators with triple graph grammars. In Mayr, E. W., Schmidt, G., and Tinhofer, G., editors, **Graph-Theoretic Concepts in Computer Science**, volume 903 of **Lecture Notes in Computer Science**, pages 151–163. Springer Berlin Heidelberg.

Schürr, A. and Klar, F. (2008). 15 years of triple graph grammars. In Ehrig, H., Heckel, R., Rozenberg, G., and Taentzer, G., editors, **Graph Transformations**, volume 5214 of **Lecture Notes in Computer Science**, pages 411–425. Springer Berlin Heidelberg.

Schwaber, K. and Beedle, M. (2001). **Agile Software Development with Scrum**. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition.

Sen, S., Moha, N., Baudry, B., and Jézéquel, J.-M. (2009). Meta-model pruning. In **Proc. of the 12th Int'l Conf. on Model Driven Engineering Languages and Systems**, MODELS '09, pages 32–46, Berlin, Heidelberg. Springer-Verlag.

Shadish, W. R., Cook, T. D., and Campbell, D. T. (2001). **Experimental and Quasi-Experimental Designs for Generalized Causal Inference**. Houghton Mifflin Company.

Shaw, M., DeLine, R., Klein, D. V., Ross, T. L., Young, D. M., and Zelesnik, G. (1995). Abstractions for software architecture and tools to support them. **IEEE Trans. Software Eng.**, 21(4):314–335.

Shaw, M. and Garlan, D. (1995). Formulations and formalisms in software architecture. In van Leeuwen, J., editor, **Computer Science Today: Recent Trends and Developments**, volume 1000 of **Lecture Notes in Computer Science**, pages 307–323. Springer Verlag.

Shlaer, S. and Mellor, S. J. (1992). **Object Lifecycles: Modeling the World in States**. Yourdon Press, Upper Saddle River, NJ, USA.

Shum, S. B. (1996). Analysing the usability of a design rationale notation. In Moran, T. and Carroll, J., editors, **Design Rationale: Concepts, Techniques and Use**, pages 185–215. Lawrence Erlbaum Associates.

Shum, S. J. B., Selvin, A. M., Sierhuis, M., Conklin, J., Haley, C. B., and Nuseibeh, B. (2006). Hypermedia support for argumentation-based rationale. In Dutoit, A. H., McCall, R., Mistrík, I., and Paech, B., editors, **Rationale Management in Software Engineering**, pages 111–132. Springer Berlin Heidelberg.

Sjøberg, D., Hannay, J., Hansen, O., Kampenes, V., Karahasanović, A., Liborg, N.-K., and Rekdal, A. (2005). A survey of controlled experiments in software engineering. **Software Engineering, IEEE Transactions on**, 31(9):733–753.

Society of Automotive Engineers (2006). SAE Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex A: Graphical AADL Notation, Annex C: AADL Meta-Model and Interchange Formats, Annex D: Language Compliance and Application Program Interface Annex E: Error Model Annex. Standard number AS5506/1.

Society of Automotive Engineers (2011). SAE Architecture Analysis and Design Language (AADL) Annex Volume 2. Standard number AS5506/2.

Society of Automotive Engineers (2012a). Architecture Analysis & Design Language (AADL). Standard number AS5506 Revision: B.

Society of Automotive Engineers (2012b). SAE Architecture Analysis and Design Language (AADL) Requirements Definition and Analysis Language (RDAL) Annex. Standard number AS5506/3 (draft.

Solberg, A., France, R., and Reddy, R. (2005). Navigating the metamuddle. In **Proc. of the 4th Workshop in Software Model Engineering**, MODELS Workshop.

Song, D., Dong, Y., Zhang, F., Huo, H., and Gu, B. (2012). Study of safety analysis and assessment methodology for aadl model. In **Software Security and Reliability Companion (SERE-C), 2012 IEEE Sixth International Conference on**, pages 174–183.

Stapleton, J. (1997). **DSDM, Dynamic Systems Development Method: The Method in Practice**. Addison-Wesley.

Takeuchi, H. and Nonaka, I. (1986). The new new product development game. **Harvard Business Review**.

Tamura, G. and Cleve, A. (2010). A Comparison of Taxonomies for Model Transformation Languages. **Paradigma**, 4(1):1–14.

Tang, A., Ali Babar, M., Gorton, I., and Han, J. (2006). A survey of architecture design rationale. **J. Syst. Softw.**, 79(12):1792 – 1804.

Tang, A., Avgeriou, P., Jansen, A., Capilla, R., and Babar, A. (2010). A comparative study of architecture knowledge management tools. **Journal of Systems and Software**, 83(3):352 – 370.

Tang, A., Jin, Y., and Han, J. (2007). A rationale-based architecture model for design traceability and reasoning. **J. Syst. Softw.**, 80(6):918–934.

Taylor, R. N., Medvidovic, N., and Dashofy, E. M. (2009). **Software Architecture: Foundations, Theory, and Practice**. John Wiley & Sons, Inc.

Terzakis, J. (2013). Ears: The easy approach to requirements syntax, version 1.0 (tutorial). In **8th Int'l Multi-Conference on Computing in the Global Information Technology**.

Tisi, M., Jouault, F., Fraternali, P., Ceri, S., and Bézivin, J. (2009). On the use of higher-order model transformations. In **Proc. of the 5th European Conference on Model Driven Architecture - Foundations and Applications**, ECMDA-FA '09, pages 18–33, Berlin, Heidelberg. Springer-Verlag.

Tratt, L. (2005). Model transformations and tool integration. **Software & Systems Modeling**, 4(2):112–122.

Tratt, L. (2006). The mt model transformation language. In **ACM Symposium on Applied computing (SAC 06)**, pages 1296–1303, New York, NY, USA. ACM Press.

Tyree, J. and Akerman, A. (2005). Architecture decisions: Demystifying architecture. **IEEE Software**, 22:19–27.

van Heesch, U., Avgeriou, P., and Hilliard, R. (2012). A documentation framework for architecture decisions. **J. Syst. Softw.**, 85(4):795 – 820.

van Heesch, U., Avgeriou, P., and Tang, A. (2013). Does decision documentation help junior designers rationalize their decisions? a comparative multiple-case study. **J. Syst. Softw.**, 86(6):1545 – 1565.

van Lamsweerde, A. (2001). Goal-oriented requirements engineering: a guided tour. In **Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on**, pages 249–262.

van Lamsweersde, A., Dardenne, A., Delcourt, B., and Dubisy, F. (1991). The kaos project: Knowledge acquisition in automated specification of software. In **AAAI Spring Symposium Series**, pages 59–62, Stanford University. American Association for Artificial Intelligence.

Varró, D. (2006). Model transformation by example. In Nierstrasz, O., Whittle, J., Harel, D., and Reggio, G., editors, **Model Driven Engineering Languages and Systems**, volume 4199 of **Lecture Notes in Computer Science**, pages 410–424. Springer Berlin Heidelberg.

Velasquez, D. and Weiss, M. (2006). Goal-oriented design of business models and software architectures. In **Canadian Conference on Electrical and Computer Engineering**.

Vestal, S. (1996). Metah programmer's manual. Honeywell Technology Center.

W3C (2004). Document object model (dom) level 3 core specification. W3C Recommendation 07 April 2004.

W3C (2007). Xsl transformations (xslt) version 2.0. W3C Recommendation 23 January 2007.

W3C (2012a). W3c xml schema definition language (xsd) 1.1 part 1: Structures. W3C Recommendation 5 April 2012.

W3C (2012b). W3c xml schema definition language (xsd) 1.1 part 2: Datatypes. W3C Recommendation 5 April 2012.

Wang, Y., Ma, D., Zhao, Y., Zou, L., and Zhao, X. (2011). An aadl-based modeling method for arinc653-based avionics software. In **Proceedings of the 2011 IEEE 35th Annual Computer Software and Applications Conference**, COMPSAC '11, pages 224–229, Washington, DC, USA. IEEE Computer Society.

Watkins, R. and Neal, M. (1994). Why and How of Requirements Tracing. **IEEE Software**, 11:104–106.

Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schoenboeck, J., and Schwinger, W. (2010). Towards an expressivity benchmark for mappings based on a systematic classification of heterogeneities. In **Proc. of the First Int'l Workshop on Model-Driven Interoperability**, MDI '10, pages 32–41, New York, NY, USA. ACM.

Wimmer, M., Strommer, M., Kargl, H., and Kramler, G. (2007). Towards model transformation generation by-example. In **Proc of the 40th Annual Hawaii Int'l Conf on System Sciences**, HICSS 2007, pages 285b–285b.

Winkler, S. and Pilgrim, J. (2010). A survey of traceability in requirements engineering and model-driven development. **Software & Systems Modeling**, 9(4):529–565.

Wirth, N. (1971). Program development by stepwise refinement. **Commun. ACM**, 14(4):221–227.

Wirth, N. (1977). What can we do about the unnecessary diversity of notation for syntactic definitions? **Commun. ACM**, 20(11):822–823.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). **Experimentation in Software Engineering**. Springer-Verlag.

Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R., and Wood, B. (2006). Attribute-driven design (ADD), version 2.0. Technical Report CMU/SEI-2006-TR-023, SEI, Carnegie Mellon University.

Woodcock, J., Larsen, P. G., Bicarregui, J., and Fitzgerald, J. (2009). Formal methods: Practice and experience. **ACM Comput. Surv.**, 41(4):19:1–19:36.

Yu, E. (1997). Towards modeling and reasoning support for early-phase requirements engineering. In **Proc. of the 3rd IEEE Int. Symposium on Requirements Engineering (RE 97)**, page 226, Washington, DC, USA. IEEE Computer Society.

Yu, E. S. K. and Mylopoulos, J. (1994). Understanding “why” in software process modelling, analysis, and design. In **Proceedings of the 16th International Conference on Software Engineering**, ICSE '94, pages 159–168, Los Alamitos, CA, USA. IEEE Computer Society Press.

Zachman, J. A. (1987). A framework for information systems architecture. **IBM Systems Journal**, 26(3):276–292.

Zdun, U. (2009). Guest editor's introduction: Capturing design knowledge. **IEEE Software**, 26(2):25–27.

Zhang, C. and Budgen, D. (2012). What do we know about the effectiveness of software design patterns? **Software Engineering, IEEE Transactions on**, 38(5):1213–1231.

Zimmermann, O., Gschwind, T., K"uster, J., Leymann, F., and Schuster, N. (2007). Reusable architectural decision models for enterprise application development. In Overhage, S., Szyperski, C. A., Reussner, R., and Stafford, J. A., editors, **Software Architectures, Components, and Applications**, volume 4880 of **Lecture Notes in Computer Science**, pages 15–32. Springer Berlin Heidelberg.

Zimmermann, O., Koehler, J., Leymann, F., Polley, R., and Schuster, N. (2009). Managing architectural decision models with dependency relations, integrity constraints, and production rules. **J. Syst. Softw.**, 82(8):1249–1267.

Zloof, M. M. (1975). Query-by-example: The invocation and definition of tables and forms. In **Proc. of the 1st Int'l Conf. on Very Large Data Bases**, VLDB '75, pages 1–24, New York, NY, USA. ACM.

Zurcher, F. W. and Randell, B. (1968). Iterative multi-level modeling - a methodology for computer system design. In **Proc. of IFIP World Computer Congress**, pages 867–871. IEEE CS Press.